



Magento[®]

eCommerce Platform for Growth

Designer's Guide to Magento

© Copyright 2011 Magento, Inc.

All rights reserved. No part of this Guide shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from Magento, Inc.

CONTENTS

1 OVERVIEW OF THEMING IN MAGENTO	1
SCOPE OF THIS DOCUMENT.....	1
AUDIENCE	1
PREREQUISITES.....	1
ABOUT THIS GUIDE.....	1
2 MAGENTO DESIGN CONCEPTS AND TERMINOLOGY.....	3
WEBSITES AND STORES.....	3
DESIGN PACKAGES AND THEMES.....	5
Themes.....	6
Default themes.....	7
Theme variations or non-Default themes.....	8
Magento's Theme Fallback Model	8
BLOCKS AND LAYOUTS.....	10
Blocks.....	10
WIDGETS	12
Widget Terminology	12
Widget Examples.....	12
3 PACKAGES AND THEMES IN MAGENTO'S DIRECTORY STRUCTURE.....	15
The Base package	15
The Default, Pro and Enterprise Packages	16
Custom Design Packages.....	18
Themes.....	19
4 APPLYING THEMES IN MAGENTO.....	21
WALKTHROUGH 1: CREATING AND APPLYING A THEME	21
Assigning packages and themes to the store	21
WALKTHROUGH 2: APPLYING MULTIPLE THEMES	23
HIERARCHY OF THEMES	28
Important Hierarchy Rules to Remember.....	31
Design Exceptions.....	31
5 CUSTOMIZING MAGENTO THEMES	33
CREATING THE DIRECTORY SKELETON FOR YOUR PACKAGE/THEME	33

Creating a new Package and theme(s).....	33
APPLYING YOUR NEW PACKAGE AND THEME TO YOUR WEBSITE.....	35
Disable your system cache	35
Apply Your Custom Package/theme to your Store.....	35
CUSTOMIZING USING THE SKIN FILES	36
Quick Exercises to Get You Started	36
Exercise #1: Modify the CSS.....	36
Exercise #2: Change the logo	36
CUSTOMIZING USING LAYOUT FILES.....	37
Introduction to Layouts.....	37
How Magento Layout Works.....	40
Anatomy of A Magento Layout File.....	42
Handles	42
<block>.....	42
<reference>.....	43
Rules of XML.....	44
How to find which layout file to modify.....	44
Quick Exercises to Get You Started	45
Exercise #1	45
Exercise #2	46
Customizing Using a Local Layout file (local.xml).....	47
CUSTOMIZING USING TEMPLATES	48
Exercise #1	49
<u>6 QUICK GUIDE TO BUILDING A THEME FROM SCRATCH.....</u>	<u>51</u>
ONE: Disable your system cache	51
TWO: Determine all the possibilities of structure types for your store	51
THREE: Cut up your xHTML according to functionality.....	52
FOUR: Change the layout to reflect your design.....	53
<u>7 OTHER RESOURCES.....</u>	<u>55</u>

1 OVERVIEW OF THEMING IN MAGENTO

The Designer's Guide to Magento expands your knowledge of the structural workings of Magento and the methods of designing for Magento. It teaches how to create a theme of your own with Magento. Due to Magento's extreme flexibility, it is not possible to document all the different ways in which it can be customized. For help with this, consult Magento's community forums and wiki, where you can profit from the knowledge of people with real design experience. Also, remember that Magento is a constantly evolving application, therefore, this documentation may not faithfully reflect your version of Magento. However, the concepts introduced to you will still be very helpful.

SCOPE OF THIS DOCUMENT

This document supports the following versions of Magento:

CEv1.4+, PEv1.8+, EEv1.8+

AUDIENCE

To understand this manual and undertake the tasks and best practices it describes, you must be thoroughly familiar with HTML and CSS.

PREREQUISITES

You need a working Magento installation if you would like to try the ideas presented in this manual.

ABOUT THIS GUIDE

This documentation contains chapters that can be skipped through back and forth to quickly access only the information you need most. However, because each chapter acts as a prelude to the next, we advise you to consider the documentation in the order it was written.

2 MAGENTO DESIGN CONCEPTS AND TERMINOLOGY

In order to follow along with this document, it is important that you have a good grasp of the terminology and concepts that underlie the frontend aspects of the Magento system. The terminology introduced in this chapter may well be new territory for you, so read through it thoroughly. But most importantly, do not be discouraged if you do not fully grasp the concepts immediately. This chapter merely introduces them to you all at once, and later chapters will expand on those simple definitions.

WEBSITES AND STORES

Because Magento natively supports the creation and management of multiple stores in a single Magento installation, Magento has a hierarchy of concepts that define the relationship between the individual stores in a Magento installation.

In Magento, a **website** is a collection of **stores**, which themselves are collections of **store views**. These layers, although perhaps initially confusing, provide you with powerful flexibility when setting up online businesses in Magento.

A **website** is made up of one or more stores which share the same customer information, order information and shopping cart.

Stores are collections of store views and can be setup in a variety of ways. Their main function is to provide a logical container that allows you to group related store views together in a website.

Store Views are the actual store instances in Magento. Most stores will have a single store view associated with them. But a store can also have multiple store views, which are typically used for different languages. Therefore, if you wanted to have a store displayed in English and Spanish, for example, you could create the store once and create two different store views for that store.

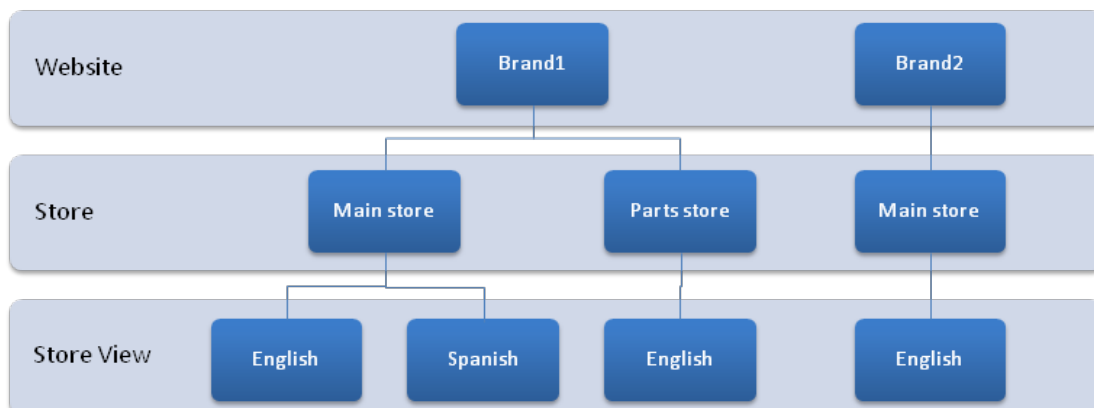


Figure 1. Hierarchy of websites, stores and store views in Magento.

These are very broad terms that can be adapted to fit the unique needs of individual merchants. A few scenarios to illustrate the different uses of websites, stores and store views follow.

SCENARIO 1—ONE STORE

A company named “Bongo’s Instruments” wants to create an online presence. Bongo has a single catalog and does not need to support multiple languages. This is the simplest scenario in which Bongo’s Instruments is the *website*, *store*, and *store view*.

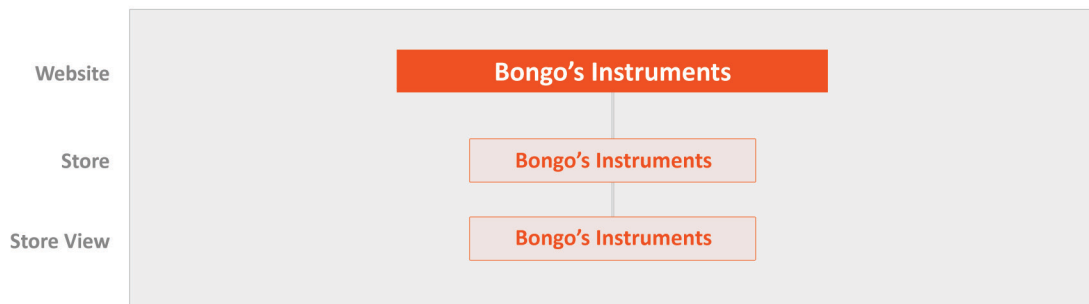


Figure 2. Single website, store and store view.

SCENARIO 2—MULTIPLE RELATED STORES

A company named “Dubloo” creates an online presence with three separate clothing stores that each cater to a different price-level audience. Dubloo wants the ability for all three of its stores to share customer and order information—meaning customers can create an account in any of the stores and it will be available to them in the others as well. In this scenario, Dubloo would have one *website* and three *stores* under their online presence. Because all of the stores support a single language, they each have only one *store view*.

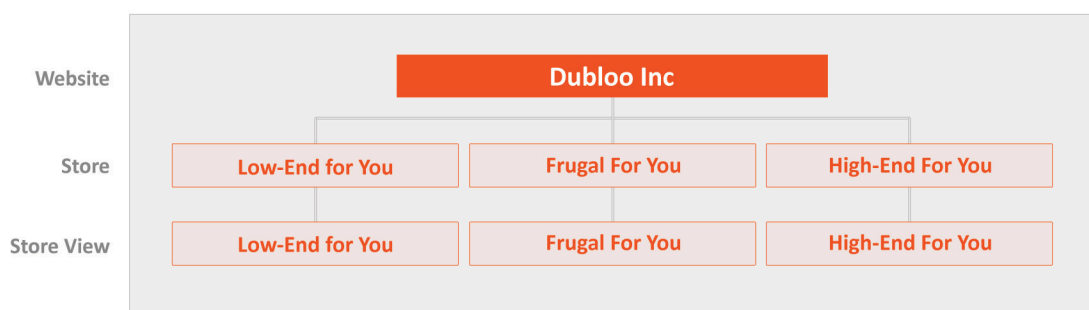


Figure 3. Single website with multiple stores.

SCENARIO 3—MULTIPLE INDEPENDENT STORES

A company named “My Laptops” wants an online presence with two separate stores that both sell laptops but at different prices and with different product selections in some categories. They also want to offer

English and Spanish language options per store. Within each store they need to synchronize customer and order information, but they do not need to share this information between the two stores. In this scenario, My Laptops would have two *websites* (which stops customer and order info from being shared with stores in the other website), each with one *store* and two *store views* (one for English and one for Spanish).

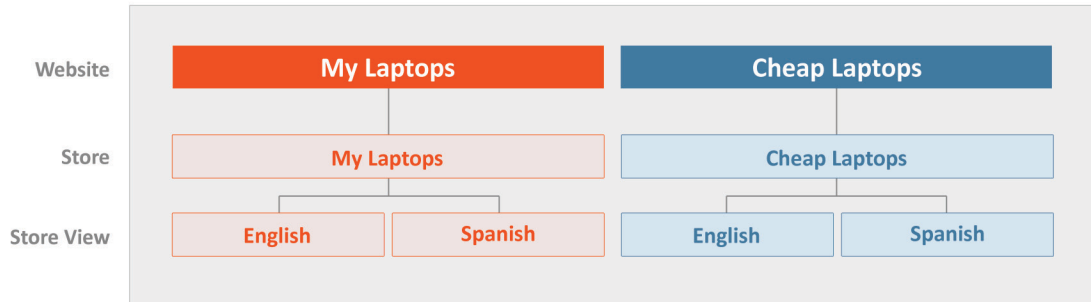


Figure 4. Multiple websites with multiple stores and store views.

DESIGN PACKAGES AND THEMES

In looking at the hierarchy of websites, stores and store views in the previous section, you can see that a single Magento installation can run many stores which may or may not share the same look and feel.

To exactly control the look and feel of each store in your Magento installation, Magento allows you to create **themes**. Related themes are grouped together in **design packages**. Each store in your Magento installation can use its own theme, or they can all share a single theme, or some combination of both.

A **design package** is a collection of related themes. If you have been working with Magento for a while, you might remember these being referred to as *interfaces* in earlier versions and earlier documentation. You can have any number of design packages installed, but there must always be at least one. When installed, Magento has a special package named the “base package.” There will also be another package specific to the Magento edition you have installed. In CE, this package is named “default;” in PE, this package is named “pro” and in EE, this package is named “enterprise.” To this you can add any number of your own custom design packages.

Themes inside of a design package contain the actual files that determine the visual output and frontend functionality of your store. Themes are covered in more detail in the next section, but briefly, Magento themes contain templating information (layout files, template files, theme-specific translation files) and skinning information (CSS files, images, and theme-specific JavaScript files). A theme can belong to only one design package. By convention, each design package must contain a **default theme**. Additionally, a design package can contain any number of variations on that default theme—called variously **non-default themes** or **theme variants**.

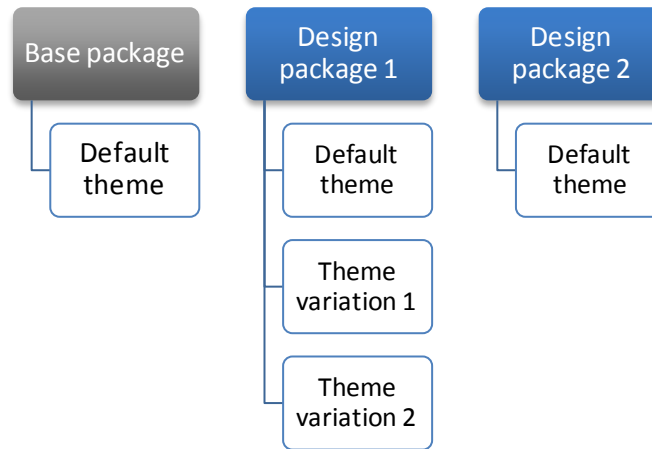


Figure 5. Relationship between design packages and themes in Magento.

Both a design package and theme can be assigned on either the website level and/or store view level through the Magento Admin Panel. If you assign a package in the website level, all your stores in that website will inherit the package of your website. You can further assign a package at the store and/or the store view level, effectively overriding that of the website.

THEMES

A Magento theme is comprised of templating files (layout, template, locale) and/or skin files (CSS, images, theme-specific JavaScript) that create the visual experience of your store. These files reside in two main directories in your Magento file system:

- “App/design” directory – Files that control how the page templates are rendered
- “Skin” directory – Files that control the visual aspects of the theme—CSS, images, etc

Magento breaks its theme files into separate directories like this to allow you more control over the security level of each directory on your server. The files in the skin directory need to be accessible to web browsers and need a very open permission setting. The files in the app/design directory only need to be accessible to the app and can be locked down further.

Inside of each of these directories, the files in a theme are broken down into further subdirectories by type of file.

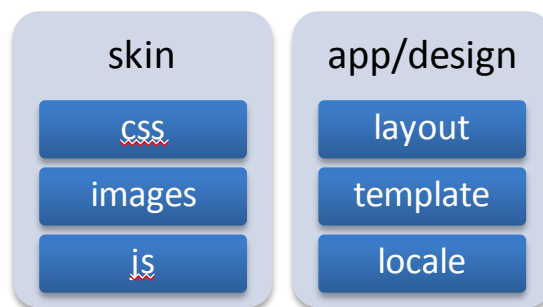


Figure 6. File types associated with Magento themes and their locations

Below is a little more explanation of each directory and the types of files it contains.

Templating files in the `app/design/frontend/<designPackageName>/<themeName>/` directory are organized into the following subdirectories:

- **Layout**—Contains the basic XML files that define block structure for different pages as well as control meta information and page encoding.
- **Template**—Contains the PHTML files that contain xHTML markups and any necessary PHP to create logic for visual presentation. Some templates are page templates and some are block templates.
- **Locale**—Contains simple CSV text documents organized on a per language basis containing translation strings (as name-value pairs) for all text produced by Magento (e.g., for interface elements and messages, not products and categories)

Skin files in the `skin/frontend/<designPackageName>/<themeName>/` directory are organized into the following subdirectories:

- **CSS**—Contains the CSS files used to control visual styling of the website
- **Images**—Contains all images used by the theme
- **JS**—Contains theme-specific JavaScript routines and callable functions. (Most JavaScript libraries, which might be shared across themes) are placed in the `js/` directory at the Magento root

DEFAULT THEMES

Every design package must include a theme named "default" which is the main theme for that package. When you assign a package to your store, unless you also specify a specific theme, Magento automatically looks for the theme named "default" in that package's directory and uses it to generate the frontend.

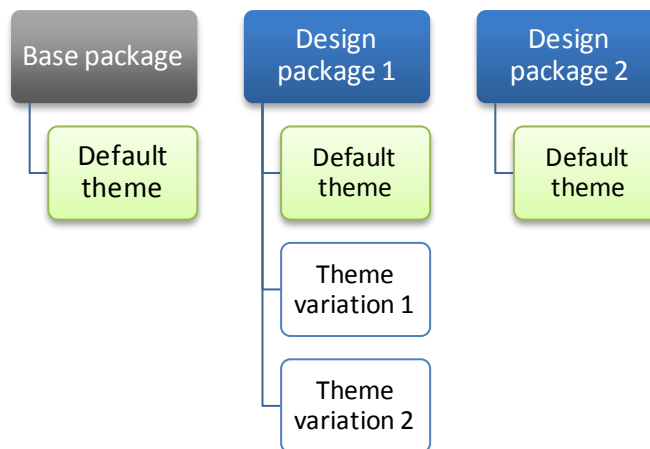


Figure 7. Design packages must have a default theme.

The default theme must contain all the required layouts, templates and skins to render your store. For any required files that are not found in the default theme of a custom package, Magento will look last in the base package's default theme.

THEME VARIATIONS OR NON-DEFAULT THEMES

A theme variation can contain as many or as few theme files as you see fit for your need. Theme variations allow you to easily create minor variations to your default theme that can be applied to your entire store, subsections of your store, specific pages in your store, or to a separate but related store. When working with a custom design package, you can either modify the default theme for that package or create another theme in that design package in addition and load it alongside the default.

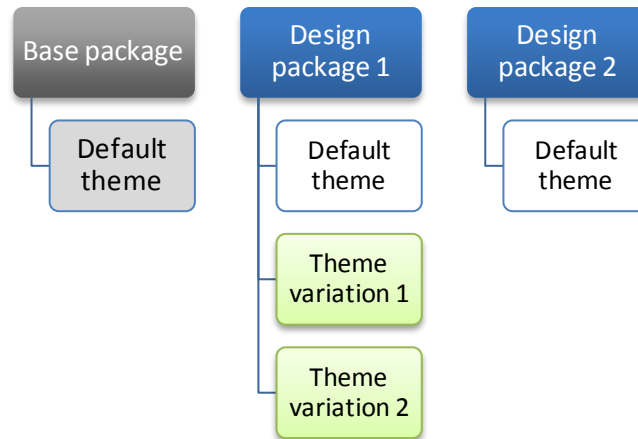


Figure 8. Theme variations or non-default themes in a design package.

Examples might include:

- A replacement parts section of your website that requires a less media-rich product page
- A section of your website that previews next season's products but does not yet offer them for sale
- A seasonal creative change that you want to apply to your entire website, but only for a limited time
- A branded website for another division of your business that needs the same overall branding as the parent site, but requires other modifications based on their catalog and business processes.

MAGENTO'S THEME FALLBACK MODEL

Magento's theme architecture was changed between Community Edition (CE) v1.3 and CE v1.4 and between Enterprise Edition (EE) v1.7 and EE v1.8 to make custom themes easier to maintain and more "upgrade proof." In these releases the theme files were also refactored and changed considerably to improve performance and accessibility, but the biggest change was to the overall theme structure and hierarchy used by Magento.

Magento has always used fallback logic in rendering themes, but with CE v1.4 and EE v1.8, this has been expanded to include the base package as the final cross-package fallback point. In previous Magento versions, fallback was only inside of a theme and customized frontend functionality could not be shared between packages without replicating all of the files to each package.

An example of how Magento's fallback logic works is, if your custom theme calls for a CSS file named "styles.css," but the Magento application is unable to find the file in your custom theme, Magento will search the next theme in the hierarchy for the file and continue searching the hierarchy of themes until it locates the file named "styles.css." This method of building design is called "fallback" because the application "falls back" to the next possible source of required files in order to retrieve and load a requested file.

The fall-back hierarchy in Magento CE v1.4+ and EE v1.8+ is shown in the diagram below.

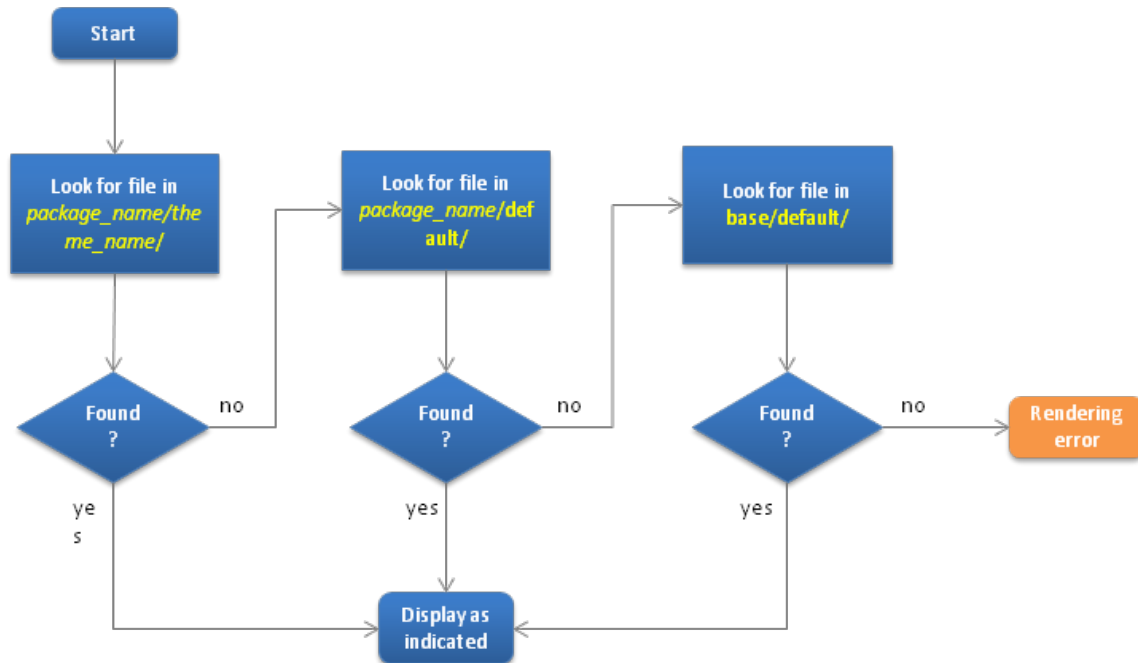


Figure 9. Logical flow of Magento's theme fallback model.

When used effectively, this fallback mechanism allows you to edit and maintain only the files you need to change as a custom theme and all of the other functionality is provided by either the custom package default theme or the base package.

For example, Magento maintains 4 CE demo themes—default, blank, modern, and iPhone. Before the introduction of the base package, all theme files had to be copied to each package and maintained, tested and debugged in each. With the introduction of the base package, we maintain three times less code. For example, the default/default and default/blank themes are implemented with just CSS changes and get all template and layout files from the base package.

Ultimately, this also makes your custom themes more "upgrade proof" and allows you to maintain less code as well. Most custom themes only customize a subset of the default Magento functionality, therefore, now all of the core functionality will be made available by the base package and can be overridden selectively by a custom package/theme.

BLOCKS AND LAYOUTS

There are some concepts and tools you need to learn to be a successful designer in Magento:

- Structural Blocks
- Content Blocks
- Layout

BLOCKS

Blocks are a way by which Magento distinguishes the array of functionalities in the system and creates a modular way to manage it from both visual and functional standpoint. There are two types of blocks and they work together to create the visual output.

Structural Blocks—These are blocks created for the sole purpose of assigning visual structure to a store page such as header, left column, main column and footer.

Content Blocks—These are blocks that produce the actual content inside each structural block. They are representations of each feature’s functionality in a page and employ template files to generate xHTML to be inserted into its parent structural block. Category list, mini cart, product tags and product listing, etc, are each its own content block. Instead of including template after template as a typical eCommerce application would in order to gather the entire xHTML output, Magento gathers and arranges page content through blocks.

Layouts—These are the files that essentially map your content blocks to your structural blocks. Layouts in Magento have two functions—they define both the structural and content blocks and then they inform Magento how and where to connect them up.

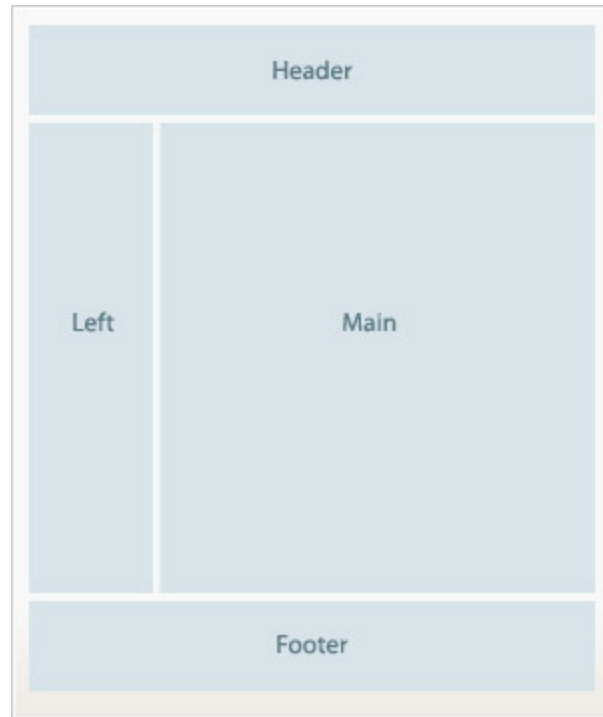


Figure 10. An example of structural blocks



Figure 11. An example of content blocks mapped to specific structural blocks

WIDGETS

You may be familiar with the term “widgets” from other online applications. In Magento, widgets are frontend blocks with a predefined set of configuration options. These configuration options are displayed in a special edit form in the backend panel when a widget is being added or edited by a store owner. The ability to easily set widget configuration options allows for the full control of widget placement on a page to store owners. An important aspect of widgets is that using them eliminates some of the theme-level customization that used to be required when setting up a Magento store.

Essentially, they are a great way to allow business users with no technical knowledge to easily add dynamic content (including product data, for example), in areas predefined by the designer/developer, to pages in Magento Stores. This allows for greater control and flexibility in creating informational and marketing content through administrator tools, enabling intuitive and efficient control of content such as:

- Dynamic product data in marketing campaign landing pages
- Dynamic Information such as recently viewed items in content pages
- Promotional images to position in different blocks, side columns and other locations throughout the storefront
- Interactive elements and action blocks (external review systems, video chats, voting and subscription forms)
- Alternative navigation elements (tag clouds, catalog image sliders)
- Interactive and dynamic Flash elements easily configured and embedded within content pages for enhanced user experience

WIDGET TERMINOLOGY

- **Frontend Block** – an element which creates the visual output either by assigning visual structure or by producing the actual content.
- **Magento Widget** – a frontend block that implements a special widget interface which allows for having different configuration options per each block instance, and the ability to have multiple independent block instances on pages.
- **Magento Widget Instance** – a block on a single page or multiple pages which receives its configuration options as defined by a store owner in the backend. The same widget can be added to the frontend multiple times producing multiple instances of the same widget.
Please note that a single widget instance can also be added to multiple pages (with the same configuration options values) and managed as a single entity.

WIDGET EXAMPLES

Magento includes the following default configurable widgets; new widgets can be created by developers as well.

- **CMS Page Link** – displays a link to a selected CMS Page, and allows specifying custom text and title. Two templates are available for this widget – inline link and block template.

- **CMS Static Block** – displays content of a selected static block.
- **Catalog Category Link** – displays a link to a selected catalog category, and allows specifying custom text and title. Two templates (inline and block) are available.
- **Catalog Product Link** – displays a link to a selected catalog product, and allows specifying custom text and title. Two templates (inline and block) are available.
- **Recently Compared Products** - displays a block which contains recently compared products. This Widget allows for specifying the number of products to be displayed and has two templates available (product list or product grid view).
- **Recently Viewed Products** - displays a block which contains recently viewed products. This Widget allows for specifying the number of products to be displayed and has two templates available (product list or product grid view).

3 PACKAGES AND THEMES IN MAGENTO'S DIRECTORY STRUCTURE

Themes are grouped together logically into design packages, but, as described in the previous chapter, the theme files are physically split between two directories. This can be a little confusing when you are first starting out with Magento theming, so this chapter shows you the directory structure associated with each of the main design packages and themes used in Magento.

As you go through this section, notice that directory names for the design packages and themes must be the same in both the `app/design` and `skin` directory. Notice also that below that level in each directory, Magento has a set of required directories in which your templating and skin files reside.

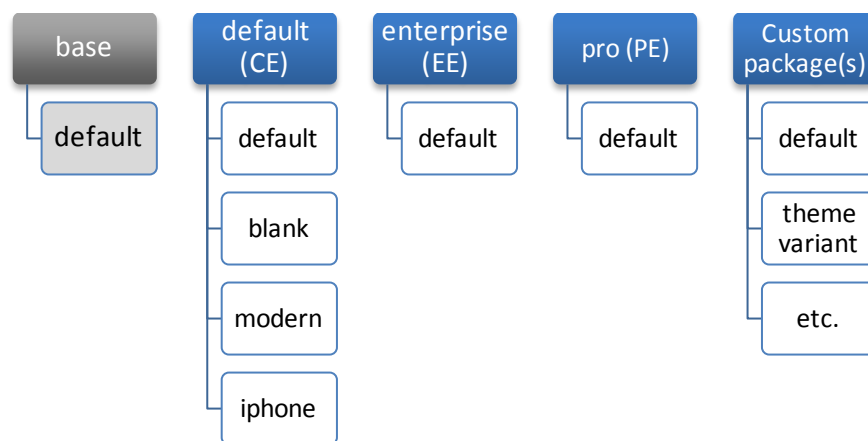


Figure 12. Package types and themes in Magento

THE BASE PACKAGE

The base package was introduced in CE v1.4 and EE v1.8. The role of the base package is to provide hooks to all of Magento's core functionality, so that your custom themes can include in them just the changes to that functionality that are specific to the design or business they are intended to support. Magento does this by using the "fallback" model in which it looks for the files it needs in order to render a page first in your custom design package and then in the base package if it does not find them.

Note: The base package is the final fallback point for all of your design packages and should never be edited or modified when you create and customize your own themes

This provides a cleaner codebase and a better upgrade path for your themes because the base package will contain all of the files that control Magento's default behavior (and there are a lot of them), but your custom theme only needs to contain the files that you have changed or that you have used to override the default behavior. The base package only contains a default theme and no other themes/theme variants should be made inside of the base package.

In the directory structure, the Base package has both templating and skin files, which are in the `app/design` and `skin` directories, respectively. The base package has only a default theme associated with it, although it is not actually a full theme because it lacks most of the skin files.

BASE PACKAGE DIRECTORY STRUCTURE (ALL MAGENTO EDITIONS)

app/design/frontend/base/default/

Contains all of the layout and template files necessary to support core Magento functionality

<MAGENTO_BASE_DIR>

```
+ app
  + code
  + design
    + adminhtml
    + frontend
      + base
        + default
      + default
    + install
  + etc
  + locale
  Mage.php
```

skin/frontend/base/default/

Contains some CSS and JavaScript files that support core functionality. It does not, however, contain all of the CSS and image files necessary to style a site, because those are specific to a design not to the core.

<MAGENTO_BASE_DIR>

```
+ skin
  + adminhtml
  + frontend
    + base
      + default
    + default
  + install
```

Base is really a repository for making core Magento functionality available to the frontend. So do not make any of your customizations in the base package—files in the base package should not be edited. Custom themes should be made inside of their own design package directories. Files that need to be changed can be copied from base to your custom design package and edited there or you can create a set of local files to hold your modifications.

Rules for working with the base package:

- Do NOT edit the files in the base package.
- Do NOT create a custom theme inside of the base package.

Note: Magento versions prior to CE v1.4 and EE v1.8 did not have a base package. Legacy instructions and tutorials on theming do not account for it, so it is important that you not follow legacy tutorials for theming that do not use the base package.

THE DEFAULT, PRO AND ENTERPRISE PACKAGES

You will notice that in every Magento installation, there is not only a base package but also at least one other design package available by default. In Community Edition (CE), this package is named “default.” In

Professional Edition (PE) this package is named “pro” and in Enterprise Edition (EE), this package is named “enterprise.”

In the directory structure, these packages have both templating and skin files, which are in the `app/design` and `skin` directories, respectively. Unlike the base package, the themes in these demo packages (`default`, `pro`, or `enterprise`) are complete themes, with full CSS files and images as well. These demo packages can be used if you want to use the default Magento theme for your edition of the platform (they look like the demo stores you see online) as your theming starting point. Their most useful purpose, however, is as a reference or example in making your own custom design package.

If you are using the Community Edition, the default package contains multiple themes and your directory structure will look like this:

DEFAULT PACKAGE DIRECTORY STRUCTURE (COMMUNITY EDITION V1.4+)

`app/design/frontend/default/`

```
<MAGENTO_BASE_DIR>
+ app
  + design
    + frontend
      + base
      + default
    + default
      + blank
      + default
      + iphone
      + modern
```

`skin/frontend/default/`

```
<MAGENTO_BASE_DIR>
+ skin
  + frontend
    + base
      + default
    + default
      + blank
      + blue
      + default
      + iphone
      + modern
```

If you are using the Enterprise Edition, the enterprise package contains just one theme and your directory structure will look like this:

ENTERPRISE PACKAGE DIRECTORY STRUCTURE (ENTERPRISE EDITION V1.8+)

`app/design/frontend/enterprise/default/`

```
<MAGENTO_BASE_DIR>
+ app
  + design
    + frontend
      + base
      + default
    + enterprise
      + default
```

`skin/frontend/enterprise/default/`

```
<MAGENTO_BASE_DIR>
+ skin
  + frontend
    + base
      + default
    + enterprise
      + default
```

Similarly, if you are using the Professional Edition, the pro package contains just one theme and your directory structure will look like this:

PRO PACKAGE DIRECTORY STRUCTURE (PROFESSIONAL EDITION V1.8+)

app/design/frontend/pro/default/

```
<MAGENTO_BASE_DIR>
+ app
  + design
    + frontend
      + base
      + default
    + pro
      + default
```

skin/frontend/pro/default/

```
<MAGENTO_BASE_DIR>
+ skin
  + frontend
    + base
    + default
  + pro
    + default
```

CUSTOM DESIGN PACKAGES

How do you create a custom design package? Most of the remainder of this document covers the steps to do this, but hopefully you can see already that based on the structure we have seen so far, you would start by making folders for your custom design package in both the `app/design` and in the `skin` directories.

For example, if you want to create a package named “Dubloo,” you would create a design package folder named “dubloo” inside of each of your frontend folders, and a “default” theme directory inside of each of those. In your theme folders, you would create the skeleton to hold the templating and skinning files you customize.

DIRECTORY STRUCTURE FOR DUBLOO CUSTOM DESIGN PACKAGE

app/design/frontend/dubloo/default/

```
<MAGENTO_BASE_DIR>
+ app
  + code
  + design
    + frontend
      + base
      + default
    + dubloo
      + default
```

skin/frontend/dubloo/default/

```
<MAGENTO_BASE_DIR>
+ skin
  + frontend
    + base
    + default
  + dubloo
    + default
```

THEMES

What goes next inside of your individual theme folders depends on the degree of customization you are doing and the Magento edition for which you are developing. Remember, because Magento can fallback to the base package to find its files, each individual theme only needs to contain the files that are different from those in the base package and those that you have added in addition to those in the base package.

As you have seen already, every design package must have at least a default theme and may have any number of theme variants. Inside of each theme, the file structure must replicate the file structure expected by Magento. Continuing with the dubloo design package example, the dubloo default theme should have at least the main theme directories as a skeleton so you have someplace to put your files as you work. All themes should have this structure, though some of these directories could be empty and can be deleted later if desired.

DIRECTORY STRUCTURE FOR THEMES IN DUBLOO CUSTOM DESIGN PACKAGE (IN CE)

app/design/frontend/dubloo/default/

```
<MAGENTO_BASE_DIR>
+ app
  + design
    + frontend
      + base
      + default
      + dubloo
        + default
          + layout
          + template
```

skin/frontend/dubloo/default/

```
<MAGENTO_BASE_DIR>
+ skin
  + frontend
    + base
    + default
    + dubloo
      + default
        + css
        + images
        + js
```

Here is where working with PE and EE are a little different than working with CE. The additional functionality in PE and EE are not part of the base package. It is made available to the frontend via the design packages, pro and enterprise, respectively. In effect, these are themselves already custom design packages. If you create a new custom package for EE or PE, you need to copy the default theme from the enterprise or pro packages as the default theme of your new design package. Then your customizations should be made as a theme variant (non-default theme) inside of your new package. Alternatively, you can create your custom theme as a theme variant inside of the enterprise or pro design package, however we recommend creating a new package in order to reduce the risk of accidentally modifying the enterprise/default or pro/default code.

DIRECTORY STRUCTURE FOR THEMES IN DUBLOO CUSTOM DESIGN PACKAGE (EE)

app/design/frontend/dubloo/your_theme/

```
<MAGENTO_BASE_DIR>
+ app
  + design
    + frontend
      + base
      + enterprise
      + dubloo
        + default (copy from EE/def)
        + your_theme
          + layout
          + template
```

skin/frontend/dubloo/your_theme/

```
<MAGENTO_BASE_DIR>
+ skin
  + frontend
    + base
    + enterprise
    + dubloo
      + default (copy from EE/def)
      + your_theme
        + css
        + images
        + js
```


4 APPLYING THEMES IN MAGENTO

In this chapter we will walk through a couple of examples of how Magento processes themes and theme files.

WALKTHROUGH 1: CREATING AND APPLYING A THEME

In this walkthrough, we will show you how to create a new design package and theme and apply it to a store in Magento. The next chapter will begin to delve into how to customize a theme. Here, assume you have been given a new package with some small customizations, either by downloading it from Magento Connect or from a colleague, perhaps.

ASSIGNING PACKAGES AND THEMES TO THE STORE

Now that you have created your own package and theme (whether a default or a theme variation), you need to assign it to your website/store for it to take effect. Navigate to the Magento admin panel (ie. www.mydomain.com/admin), then the Design configuration tab (System → Configuration → Design tab). If you have more than one website, store or store view defined in your Magento installation, in the upper corner of the left column, you will see a box labeled "Current Configuration Scope" where you can select the store to which you are applying your configuration changes.

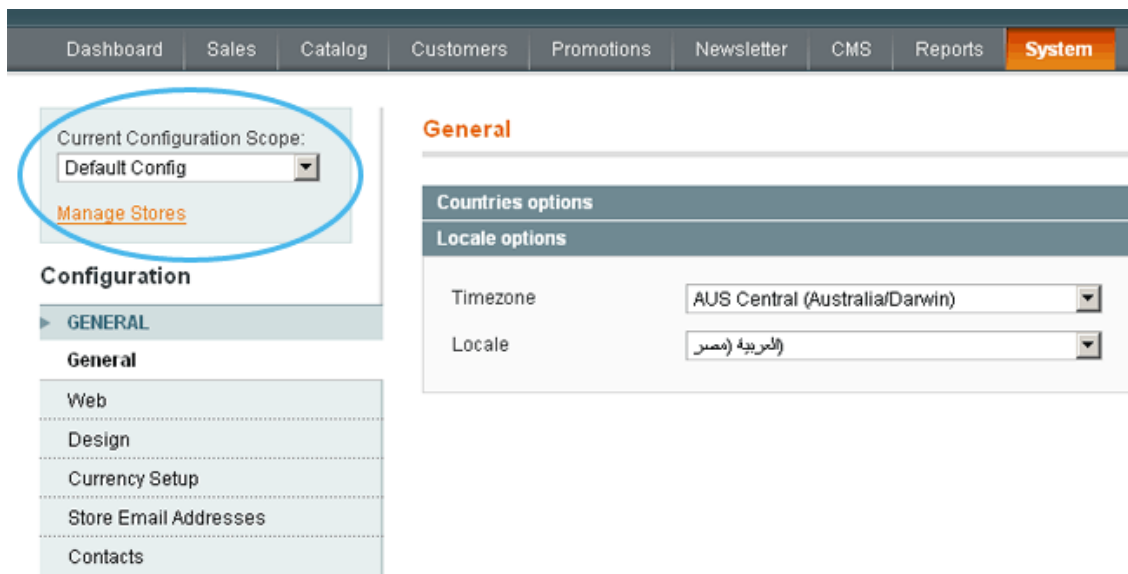


Figure 13. Setting configuration scope in the Magento Admin Panel.

In order to manage your store design at the website-level, select the name of your website from the dropdown, then apply the following steps. What do we mean by website-level? Remember Dubloo Inc?

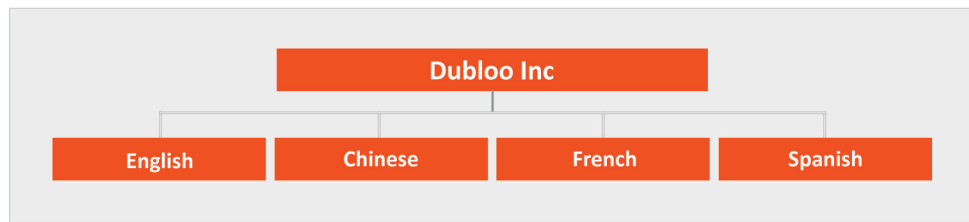


Figure 14. Dubloo example with all stores sharing the same theme.

In this case, Dubloo Inc wants each of their store views to look the same (they are all orange), so they do not need to set their design per website.

In order to manage the design from the store view-level, select the name of your store view from the dropdown, then apply the following steps. Let's look at Dubloo Inc again.

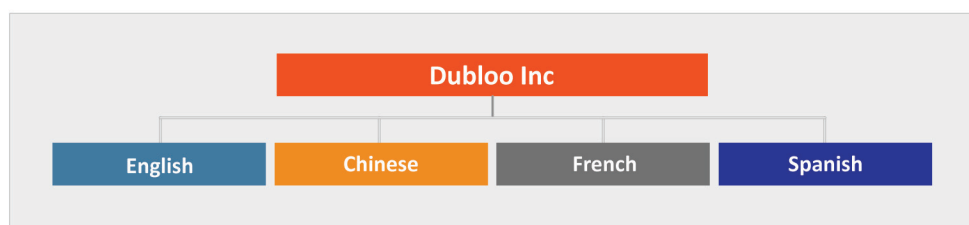


Figure 15. Dubloo example with all stores having a unique theme.

Let's say Dubloo Inc wants their users to see a visual difference depending on which store view they are using. They would need to manage the design from the store-view level to achieve this.

STEP 1

From the Design tab, in **Current package name**, enter the name of the package in which your new theme resides. Magento automatically sets the package named "default" if this box is left blank.



Figure 16. Setting the Design Package in the Magento Admin Panel.

STEP 2

In the default field (in the Themes panel), enter the name of the new theme you would like to apply to your website, store or store view. If you leave this field blank, Magento will apply the “default” theme of the design package you set in step 1. Remember, according to Magento’s fallback scheme, if you specify a theme name, Magento will look for files first in the specified theme, then in the “default” theme for your design package and finally in “base/default”. Notice that you can also override the theme you have applied for specific file types such as layouts, templates, translations and skins. If you enter a different theme name in any of these fields, Magento will look for those file types first in the theme you have indicated there and then fallback to the “default” theme for this design package.

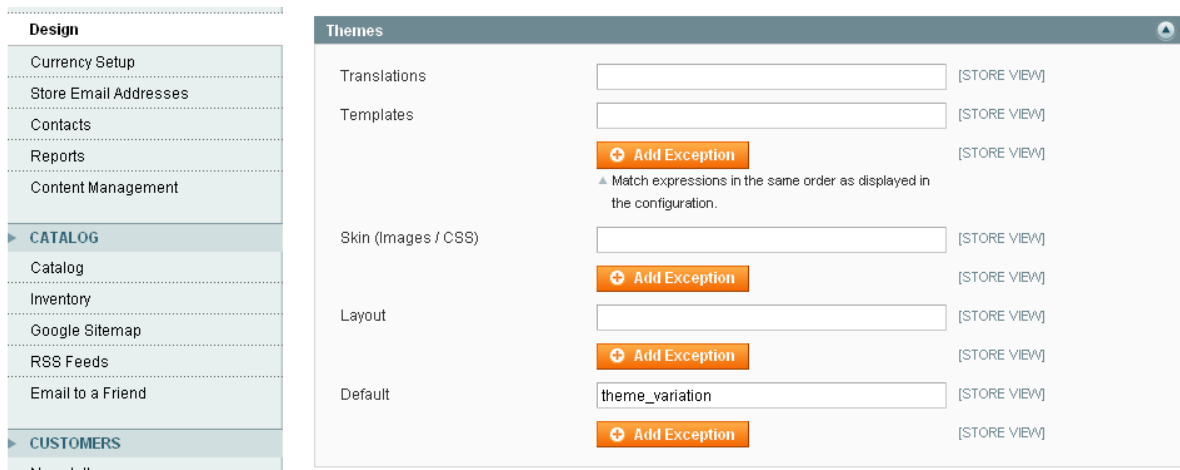


Figure 17. Setting the theme in the Magento Admin Panel.

STEP 3

When you are done, click **Save config** and reload your store. Voila! You now see your new theme reflected in the frontend.

Now that we have covered how to create and manage themes, let’s move on to how Magento handles those themes.

WALKTHROUGH 2: APPLYING MULTIPLE THEMES

Magento allows you the flexibility of using multiple themes even within the same layout and template. Let’s say you want to have the same template structure and layout across all store views but vary the actual graphics and color scheme. Magento makes this really simple.

Remember how our layout and template folders are located within `app/design/frontend/[package]/[theme]` and our actual style is located within `skin/frontend/[package]/[theme]`? The two do not have to be named the same!

If you look at the default Magento installation, you will see the default theme folder in `app/code/design/frontend/default/default`. Corresponding with this are multiple skin folders. Take a look in `skin/frontend/default` and you will see “blank,” “blue,” “default,” “French” and “german.” All five are different skins that can be used with the default theme. The “blank,” “iPhone” and “modern” themes are listed, too, but interact separately from the “default” theme. There are more themes defined in the skin directory than in the `app/design` directory, which is because the “blue,” “French” and “german” themes differ from the “default” theme only in their CSS.

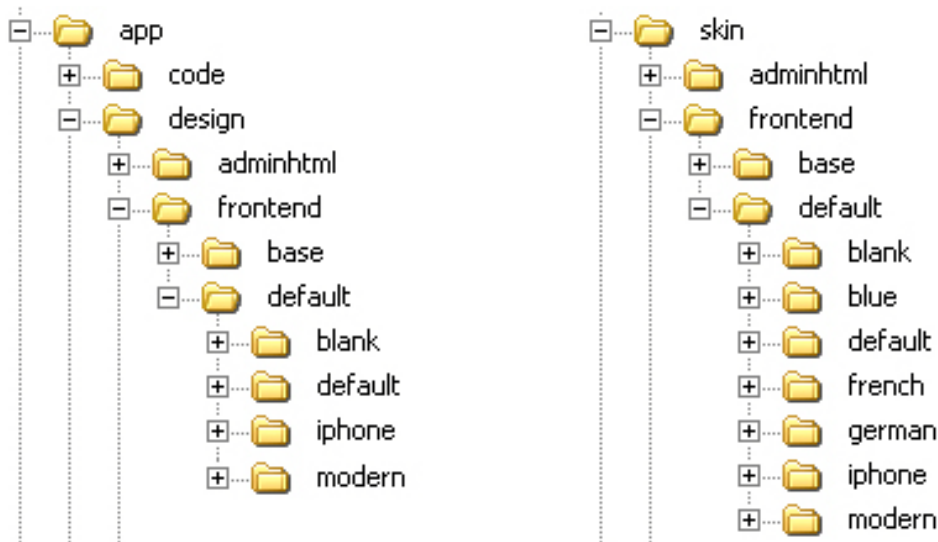


Figure 18. Screenshot of directories (themes) in CE default package.

Let’s go back to Scenario 2 from our Design Terminologies section. Remember the My Laptops store?

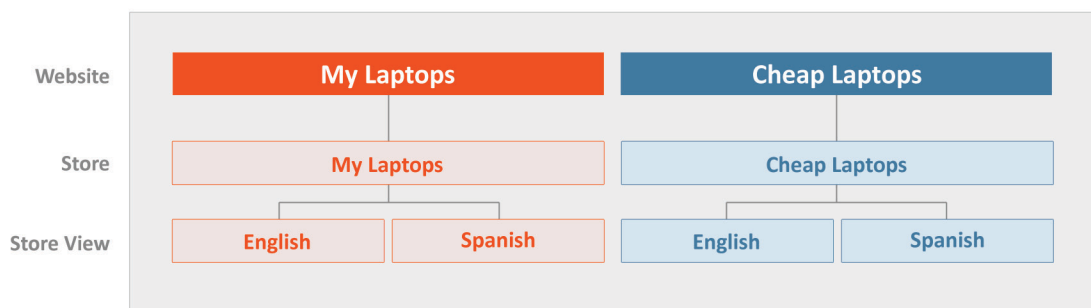


Figure 19. Laptop store with multiple websites, stores, and store views.

In this case you could choose “default” for your layout and templates. Let’s assume you want to leave your English Store View using the default skin. Because we have already set this as the Default for the entire configuration, you actually won’t need to set anything for this view.

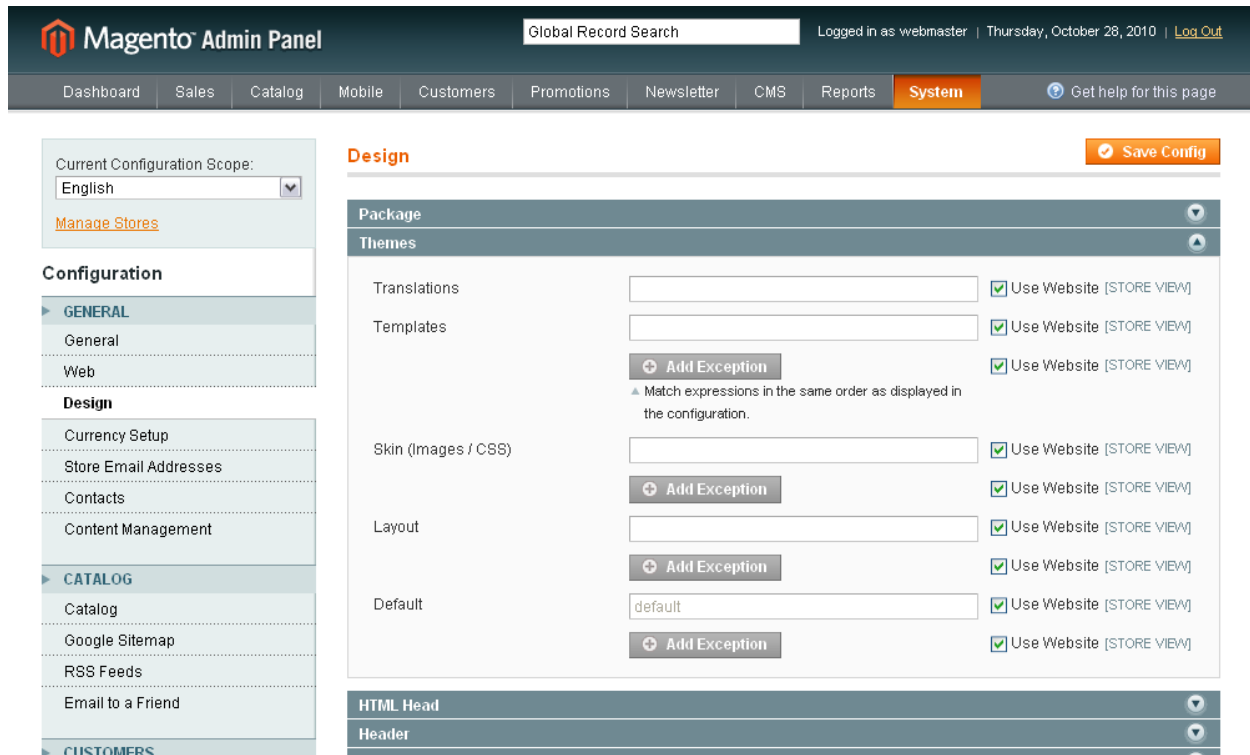


Figure 20. Setting the theme of the English store to default/default theme (CE).



Figure 21. Frontend of English store with default/default theme (CE).

Now, let's say you want your Spanish users to see a notable difference, besides just the language, when they switch store views. To do this you can assign the "blue" theme variation to your Spanish store view.

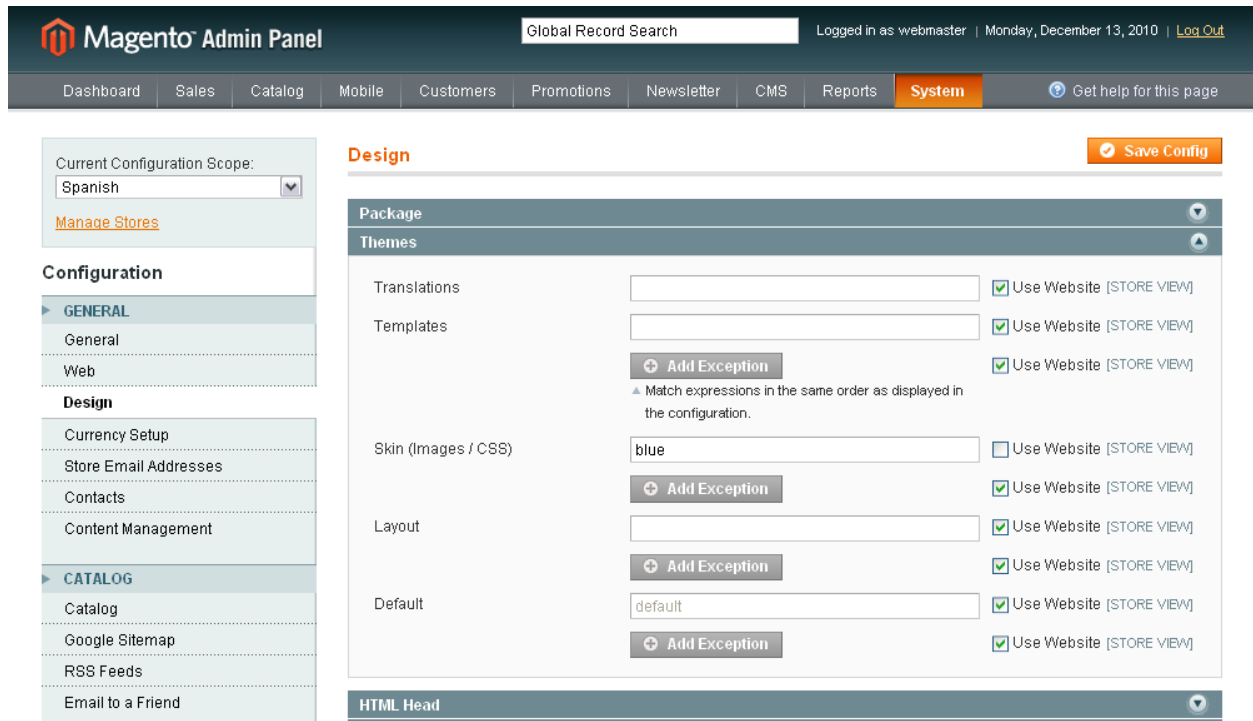


Figure 22. Setting the theme of the Spanish store to default/blue theme (CE).

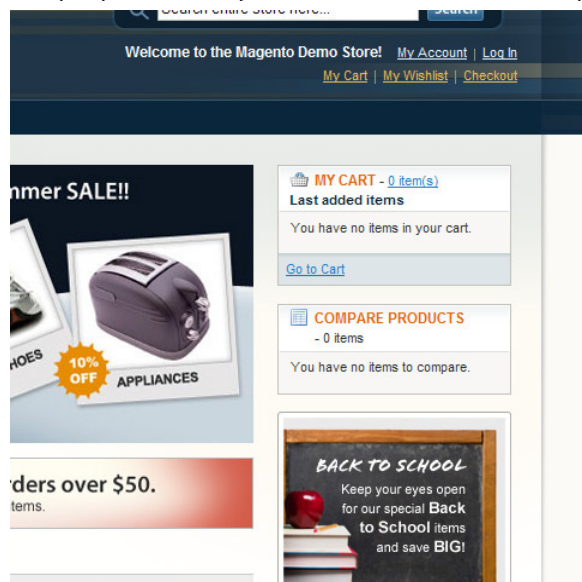
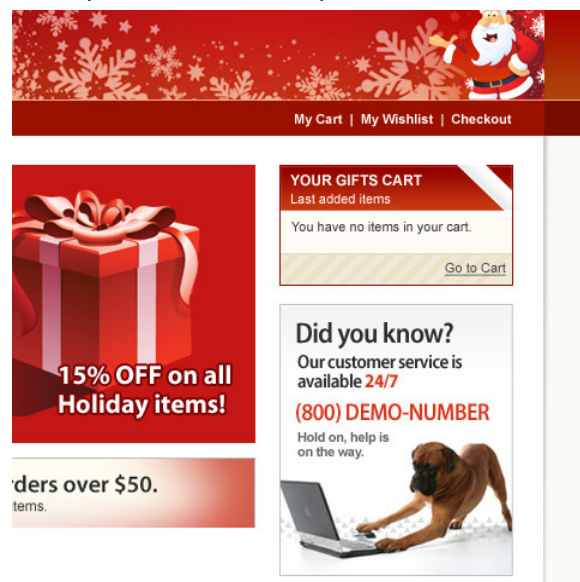
Notice that we still left the Default as "default" so that anywhere we do not have a change in our "blue" theme variation it will fallback to that value our "default" theme. This will allow you to have a second set of graphics, and color palette if you so choose, based on your target audience.

See what happens when we select Spanish on the frontend:



Figure 23. Frontend of Spanish store with default/blue theme (CE).

Let's see another example. Holiday seasons by far offer the most extensive sales opportunity for any eCommerce store. Customers line up to buy Christmas gifts for their family and friends, and moms line up to buy Halloween costumes for their child's special night of trick 'or treating. In order to tailor to the seasonal shoppers, your store must faithfully reflect the occasions in order to inspire your shoppers to explore your store. A shop like the Everyday look below just does not cut it during Christmas. What this store needs is a few reds, snowflakes and Santa Claus—just like the store below in the Holiday look!

Everyday look. This just does not cut it for holidays**Holiday look. This is what you need****Figure 24. Seasonal looks that are based on the same theme but with variations.**

Magento affords the capacity for your store to handle multiple themes of your choice exactly for those times when an extra touch is needed most. In this case, you would not need to create a second store view like we did with English and French. All you would need to do is upload a new Christmas skin to your skins folder and then reference it in your configuration where we assigned “default” and “french.” By doing this you are still preserving your normal non-seasonal store design and structure, while enhancing it with the Christmas theme. When your Christmas season is over, simply change the skin name back to your typical choice and voila!—your store is back to normal.

You will notice similarities in the underlying design of the two designs above. The facade has gone Christmas in Diagram 4, but underneath it you can still see the structure of the off-season store design. The only real differences between the two store designs are just a few CSS and image files and wording changes in the template files. Because the changes are in fact minor, you do not need a whole new default theme to accommodate your Christmas theme. What you need is just a few file replacements, and you are on your way to a much merrier store. Magento’s multiple themes functionality was created to accommodate exactly that need, putting the power at your fingertips to turn on and off the seasonal themes while preserving your default theme.

HIERARCHY OF THEMES

When you assign multiple themes to your store, you are taking advantage of the fact that although your main goal when building a theme is to create the most usable and visually pleasing graphical interface, Magento’s is to ensure that the application is able to locate and load all the files of the theme required to keep the application running error-free.

For instance, if your category listing page calls for a template named "view.phtml" (in which case this template becomes a required file), but the application is unable to find the file in the theme highest in hierarchy, it will look to the next theme highest in hierarchy to find the file. Should this fail, it will continue working down the hierarchy of themes until it is able to locate the file named "view.phtml," upon which the application will load it to the store and terminate the search. As explained earlier in this guide, this method of building design is called "fallbacks," because the application falls back to the next possible source of required files in order to retrieve and load it. See [Magento's Theme Fallback Model](#) for an introduction to fallbacks.

The fallback hierarchy in Magento CE v1.4+ and EE v1.8+ is as follows.

1. Look for requested file in:
 - app/design/frontend/design_package/theme_variation/
 - skin/frontend/design_package/theme_variation
 - Look for specific block overrides in a local.xml layout file
2. If not found, look for requested file in:
 - app/design/frontend/design_package/default
 - skin/frontend/design_package/default
3. If not found, look for requested file in:
 - app/design/frontend/base/default
 - skin/frontend/base/default
4. If not found, a rendering error will occur.

For example, let's say you have three themes assigned to your store and each of these themes contains the following files:

Table 1

base	default	theme_variation
All required files	templates/3-col-layout.phtml	templates/3-col-layout.phtml
	templates/header.phtml	css/base.css
	images/logo.gif	
	css/base.css	
	css/boxes.css	

Let's also assume that the three themes are assigned this hierarchy:

Table 2

HIGHEST	theme_variation
	default
LOWEST	base

You will see that there are few redundant files such as `templates/3-col-layout.phtml` and `css/base.css` in Table 1. Now, let's arrange the table so the redundant files are arranged parallel to one another.

Table 3

base	default	theme_variation
All required files		
	<code>templates/3-col-layout.phtml</code>	<code>templates/3-col-layout.phtml</code>
	<code>templates/header.phtml</code>	
	<code>images/logo.gif</code>	
	<code>css/base.css</code>	<code>css/base.css</code>
	<code>css/boxes.css</code>	

"Ok, great. But what does this mean?" you may ask.

Well, the files in Table 3 are how you see the files in each theme folders and not how Magento sees it.

Let's then look at how Magento sees the same file structure in Table 4.

Table 4

base	default	theme_variation
All required files		
		<code>templates/3-col-layout.phtml</code>
	<code>templates/header.phtml</code>	
	<code>images/logo.gif</code>	
		<code>css/base.css</code>
	<code>css/boxes.css</code>	

You will notice how Magento ignores the version of the redundant file lower in hierarchy and recognizes only the version higher in the hierarchy. This is because it has already found the required file and needs not search for it any longer, therefore terminating the search for that specific file and continuing the search for other required files yet to be found.

When used effectively, this fallback mechanism allows you to edit and maintain only the files you need to change as a custom theme and all of the other functionality is provided by either the custom package default theme or the base package.

For example, Magento maintains four CE demo themes—default, blank, modern, and iPhone. Before the introduction of the base package, all theme files had to be copied to each package and maintained, tested and debugged in each. The code in the base package is more streamlined.

For example, the default/default and default/blank themes are implemented with just CSS changes and inherit all of their template and layout files from the base package.

Ultimately, this will also make your custom themes more “upgrade proof” and allow you to maintain less code as well. Most custom themes only customize a subset of the default Magento functionality, therefore now all of the core functionality will be made available by the base package and can be overridden selectively by a custom package/theme.

IMPORTANT HIERARCHY RULES TO REMEMBER

- Create your customized themes inside of their own design package. Make directories at `app/design/frontend/your_design_package/default` and `skin/frontend/your_design_package/default` and build your custom theme there.
- Do not copy all the files from `base/default` into your custom package. Copy only the files that you modify. This will help you know what to look for later when custom files need updated without searching needlessly through unmodified files.

DESIGN EXCEPTIONS

Design exceptions allow store owners to specify an alternative theme for a specific user-agent. Instead of creating separate store views for user-agents (example: iPhone), design exceptions allow store owners to override design settings of a store view, thus limiting the amount of store views that need to be managed.

For example, to use the iPhone theme you can add a theme exception where the Matched Expression would be “iPhone” and the Value would be iPhone (if that’s the name of the theme you want to use).

Current Configuration Scope:
Default Config

[Manage Stores](#)

Configuration

GENERAL

General

Web

Design

Currency Setup

Store Email Addresses

Contacts

CATALOG

Catalog

Inventory

Google Sitemap

RSS Feeds

Email to a Friend

CUSTOMERS

Newsletter

Customer Configuration

Wishlist

SALES

Sales

Design

Package

Current package name: default

[+ Add Exception](#)

Match expressions in the same order as displayed in the configuration.

Themes

Translations:

Templates:

[+ Add Exception](#)

Match expressions in the same order as displayed in the configuration.

Skin (Images / CSS):

[+ Add Exception](#)

Layout:

[+ Add Exception](#)

Default:

Matched expression	Value	
iPhone	iphone	Delete
iPod	iphone	Delete
		+ Add Exception

Figure 25. Applying design exceptions allows you to trigger Magento to use a different theme for different clients/browsers.

5 CUSTOMIZING MAGENTO THEMES

Magento is built on a fully modular model that provides you with nearly unlimited flexibility. By nature of the application, this approach to programming is mirrored in the way you will develop themes for your store. In this chapter, we will explore what that means for you and exactly how to go about developing a theme for your store with Magento.

This chapter helps you understand how to modify an existing template. From there you will gain the comfort and knowledge to develop your own. Do not forget that you are still building off the base package rather than completely from scratch. Our ultimate goal is that you be able to customize your store to look and behave exactly as you want it to, but that your theme directories will contain the minimum number of files to make that happen and rely on the base theme for everything else.

CREATING THE DIRECTORY SKELETON FOR YOUR PACKAGE/THEME

With the inclusion of the base package, creating a new design package and default theme has never been easier. You can create a new theme by including just the files that need to be different from the base. One rule you must always remember to follow is to make certain that you preserve the subdirectory structural conventions of Magento. A good starting point when creating a custom design package and theme is to create an empty “skeleton” reflecting all of the main templating and theming directories used by Magento so that you can easily save your files to the correct locations. When you are done, if any of these directories remains empty and were unneeded, you can delete them then if desired.

CREATING A NEW PACKAGE AND THEME(S)

Please ignore legacy Magento instructions and tutorials that instruct you to create your custom theme inside of the default design package, or to edit files in the default/default directory directly. Rather, the method that affords the best upgrade path for your theme and the most protection from accidental changes is to create a new design package and to create your custom theme inside of there.

Inside of app/design and skin, create a new design package directory and in each create the templating and skinning directories required by Magento as shown below.

DIRECTORY STRUCTURE FOR THEMES IN DUBLOO CUSTOM DESIGN PACKAGE (CE)

app/design/frontend/dubloo/default/	skin/frontend/dubloo/default/
<pre><MAGENTO_BASE_DIR> + app + design + frontend + base + default + your_package + default + layout + template</pre>	<pre><MAGENTO_BASE_DIR> + skin + frontend + base + default + your_package + default + css + images + js</pre>

If you are working in CE, this is all you need to do to have things set up correctly for you to get started—Magento will look for files here in the default theme for your custom design package and then fallback to the base package if it does not find them. Because the base directory does not have full CSS, it is often a good idea to copy the skin files from one of the default design package themes as a starting point. The default/blank theme CSS and images are the leanest.

If you are working with EE or PE, please remember that the EE and PE enhanced functionality is itself implemented as a design package. When customizing EE or PE, you can create your custom theme as a variant inside of the enterprise or pro design package, but we suggest to create a new design package and then to copy over the default theme from EE or PE into the new design package as its default theme. This protects the original enterprise or pro themes from being accidentally edited and you can always go back to them to get fresh files if you have made an error. In addition, when you upgrade Magento, the default theme in each of the enterprise and pro packages will be updated automatically as well, which may break your custom theme before you have had a chance to test it.

Inside of app/design and skin, create a new design package directory and in each copy the default theme from enterprise or pro to here as the default theme. Then create directories for a theme variant in each and create the templating and skinning directories required by Magento as shown below.

DIRECTORY STRUCTURE FOR THEMES IN DUBLOO CUSTOM DESIGN PACKAGE (EE)

app/design/frontend/dubloo/my-theme/	skin/frontend/dubloo/my-theme/
<MAGENTO_BASE_DIR>	<MAGENTO_BASE_DIR>
+ app	+ skin
+ design	+ frontend
+ frontend	+ base
+ base	+ enterprise
+ enterprise	+ your_package
+ your_package	+ default (copy from EE/def)
+ default (copy from EE/def)	+ your_theme
+ your_theme	+ css
+ layout	+ images
+ template	+ js

With the structure above, Magento will look for files in the my_theme theme in your custom design package, then in the default theme for your custom design package and then fallback to the base package if it does not find them.

APPLYING YOUR NEW PACKAGE AND THEME TO YOUR WEBSITE

Once you have created the skeleton directory structure to hold your new design package and theme, you will need to apply it to your Magento store so that you can see changes as you work.

DISABLE YOUR SYSTEM CACHE

To prepare your Magento for production, you need to first disable system cache by going to the Administration Panel (<http://yourhost.com/admin>) and navigating to System → Cache Management. Select the check boxes next to **Layouts**, **Blocks HTML output** and **Translations** and then under Actions select **Disable** and click **Submit**. Each of those Cache Types should have a red bar in the status area that reads "DISABLED." This helps to ensure that you see a faithful reflection of your store front as you make the changes.

Note: Depending on what you are attempting to achieve, you may need to disable additional cache files in this area or even manually delete cache files in your folder structure. Do not manually delete folders unless you know what you are doing.

APPLY YOUR CUSTOM PACKAGE/THEME TO YOUR STORE

To see your new theme as you develop it, be sure to apply it to your development website using the Magento Admin Panel. Go to the ADMIN: System > Configuration > Design tab. In the Package panel, set the "Current Package Name" to whatever you have named <my_package>. In the Themes panel, set the "Default" theme to default if you are creating a new theme in CE or to whatever you have named <my_theme> if you are in EE or PE.

If you go back to the frontend of your website and refresh the browser to reload the page, you will see your store. It should have all the default functionality available from the base package and will have whatever CSS you have moved into your theme controlling the styling. If you did not move any starting CSS and images into your theme, the site will show text-only, unstyled content.

CUSTOMIZING USING THE SKIN FILES

Manipulation images and CSS can customize a Magento theme very effectively and efficiently. If you recall from an earlier example, the blue theme in Community Edition differs from the default theme only in the CSS. Do not use CSS visibility to turn on and off Magento blocks and content; use it to do all of your site styling.

Because Magento has so much built-in functionality wrapped in pre-defined divs, until you are fairly experienced with theming in Magento, it is often best to start with existing style sheets and modify them to fit your branding and look. When you are more experienced with Magento, you can start from scratch or layer in your favorite framework. If you do start with existing CSS and images, be sure to download and install a tool like the Firebug plug-in for Firefox, which will help you see which CSS is controlling the formatting of every element on the page.

QUICK EXERCISES TO GET YOU STARTED

EXERCISE #1: MODIFY THE CSS

1. Use Firebug to explore the existing CSS that's being applied to the website.
2. Make changes to colors, padding, etc. in the main CSS file for your theme. Remember to change only the file inside of your custom package and theme. If the main CSS file is named "styles.css," you will be changing the file at this location:
`skin/frontend/<your_package>/<your_theme>/css/styles.css`
3. Go to the frontend and reload the page to see your changes as you work.

EXERCISE #2: CHANGE THE LOGO

Certainly one of the first things everyone wants to do when customizing a Magento store is to change the logo. The logo for the store is associated with the theme that will be used for that store and is stored in the `skin/images` directory for that store. By convention, Magento uses 3 logo files named:

- **logo.gif**—used in the header area of the website itself
- **logo_email.gif**—used in outbound emails from the website
- **logo_print.gif**—used when printing the website

If you look at the image directory in an existing theme's skin directory, such as Community Edition's "default/blank" theme, in the `skin/frontend/default/blank/images` directory, you will see these three files.

You can change the logo on a website using your theme simply by putting your own logo files into your custom theme skin directory with the same name, if desired.

1. Upload your logo files with the same names to the `skin/frontend/<your_package>/<your_theme>/images` directory

You can change the name and file type of the logo file by changing the filename in the Admin Panel:

1. Go to ADMIN: System> Configuration> Design tab. In the Header panel, set the path and name of the main logo file. This only sets the logo file that's used on the website, not the `_email` and `_print` versions.
2. Upload your main logo file with the new name to the `skin/frontend/<your_package>/<your_theme>/images` directory

CUSTOMIZING USING LAYOUT FILES

Layouts are one of Magento's most powerful and most unique features for theming. Even if you are familiar with other templating systems, working with layouts is likely to be new to you and may at first feel like just a confusing additional step. If you master Magento's use of layouts, you can quickly turn on, turn off, and move about almost all of Magento's content and functional blocks.

INTRODUCTION TO LAYOUTS

Layout is the tool with which you can assign content blocks to each structural block you create. Magento's layout files are in the form of XML text-files and by modifying the layout you are able to move blocks around in a page and assign templates to the content blocks to produce markup for the structural blocks. In fact, with the help of a few layout files alone, you are able to modify the visual layout of every page in your store.

As an example, look at this screen shot of the category landing page in Magento:

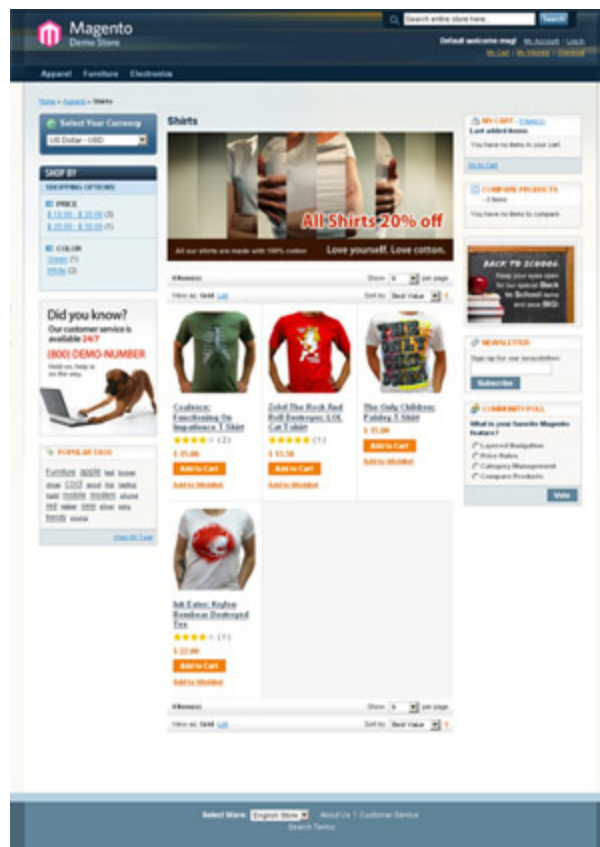
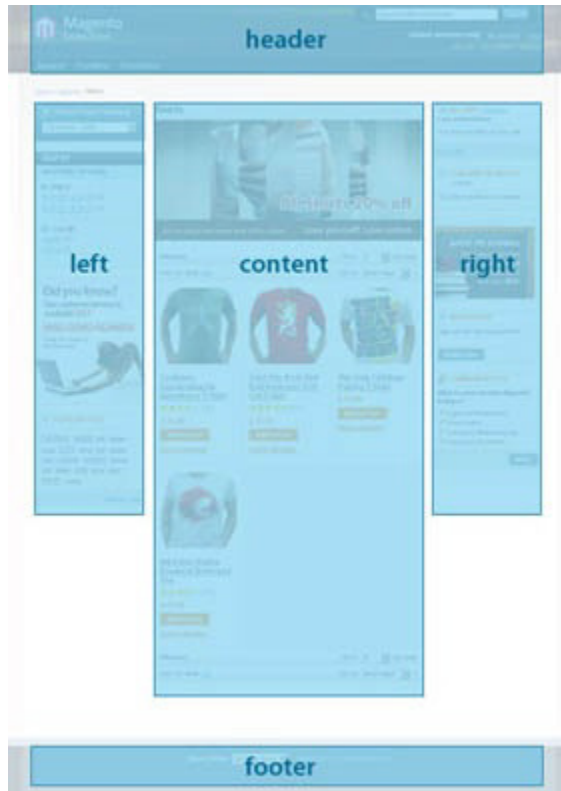
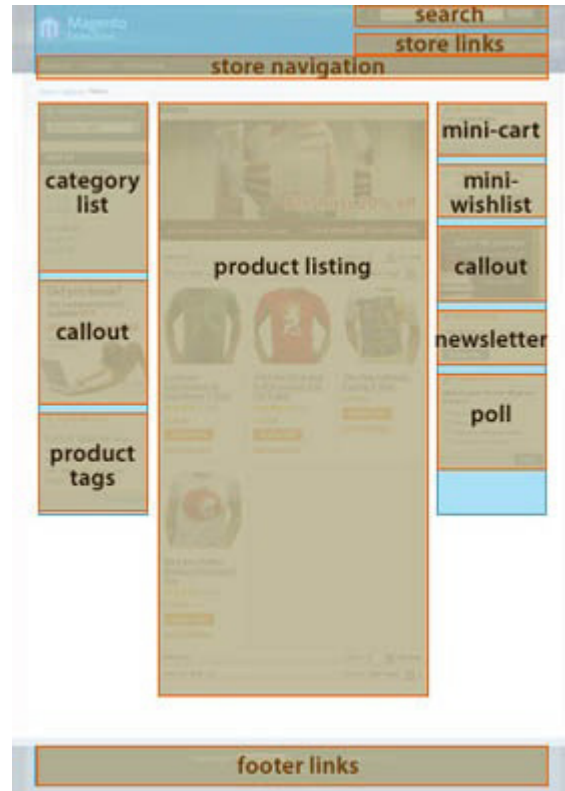


Figure 26. A sample category page from the Magento sample store.

Now look at the break-down of the above snapshot into Magento's structural blocks and the content blocks assigned to each of those structural blocks.

Magento's default structural blocks:**Content blocks mapped to structural blocks:****Figure 27. Overlay showing Magento's use of structural and content blocks.**

In concept, the outlines in the left diagram above are the **structural blocks**. They are the parent blocks of content blocks and in realization, serve to position its content blocks within a store page context (as in the right diagram above). These structural blocks exist in the forms of the header area, left column area, right column, etc., which serve to create the visual structure for a store page. In this chapter, we're going to work with Magento's default structural blocks. Structural blocks are handled by Magento's Page module and are defined in `app/design/frontend/base/default/layout/page.xml` file and the default page templates defined in the `app/design/frontend/base/default/template/page/` directory.

A **content block** is the individual-colored blocks that make up a structural block. In a store context, they are the true content of a store page. They are representations of each functionality featured in a page (such as category list, callout and product tags, etc.), and employ template files to generate xHTML to be inserted into its parent structural block. Content blocks make up the bulk of the remainder of the template files contained in the `app/design/frontend/base/default/template/` directory.

Layout is the tool with which you can assign content blocks to each structural block you create. Layout exists in the form of XML text-file and by modifying the layout you are able to move blocks around in a page and assign templates to the content blocks to produce markup for the structural blocks. In fact, with the help of a few layout files alone, you are able to modify the visual layout of every page in your store.

With Magento, you will no longer have a template file named “left_column.ext” and in it have the never-ending morass of markups that must be manually managed depending on each functionality needed for the page. Instead, your templates are managed on a per-functionality basis and you can load and unload functionalities in your store page by the virtue of a couple of layout commands alone.

HOW MAGENTO LAYOUT WORKS

Layout is a virtual component of the Magento application. By modifying the components of layout, you can build your store page in an upgrade-compatible way. The layout files at first appear complex, but in actuality, they are built using a small set of XML tags that are easy learn. By learning some key concepts and commands of layout, you will soon be armed with the confidence and knowledge to easily modify your store design to your desired specifications.

The base layout files are found in the directory: `app/design/frontend/base/default/layout`.

Layout files for a theme should be placed in the directory:
`app/design/frontend/<your_package>/<your_theme>/layout`

A layout is comprised of default layout and layout updates that are made up of easy-to-learn XML tags. With these layout commands, you can modify/assign content block-structural block relationships and also control store-front functionalities such as loading and unloading of block-specific JavaScripts to a page.

Each Magento module has its own layout file, for instance “catalog.xml” is a layout file for the catalog module” and “customer.xml” is for the customer module. Each file is further separated into **handles**. The handles that are available are pre-defined in the Magento core and by any active extensions in your store. For the most part, each handle corresponds to a type of page within your Magento store, such as `<catalog_category_default>` and `<catalog_product_view>`, but there are some handles that apply to all pages, such as the `<default>` handle, and some that apply within pages based on specific statuses, such as `<customer_logged_in>` and `<customer_logged_out>`.

```

<default> → Handle
<!-- Mage_Catalog -->
<reference name="top.menu">
    <block type="catalog/navigation"
</reference>
<reference name="left">
    <block type="core/template" name
        <action method="setImgSrc"><
        <action method="setImgAlt"><
        <action method="setLinkUrl"><
        <action method="setLinkUrl">
    </block>
</reference>
<reference name="right">
    <block type="core/template" name
    <block type="core/template" name
</reference>
</default>

<!--
Category.default.layout
-->

<catalog_category_default> → Handle
<reference name="left">
    <block type="catalog/navigation"
</reference>
<reference name="content">
    <block type="catalog/category_view"
        <block type="catalog/product"
    </block>
</reference>
</catalog_category_default>

```

Figure 28. Code sample from Magento layout files showing use of handles

A handle can occur in multiple layout files and blocks can be assigned to that handle or overridden by each layout file. For example, most layout files may contain the `<default>` handle. When parsing the layout files, Magento first grabs the layout updates assigned in the `<default>` handle in each of the layout files, reading them in the order as assigned in `app/etc/modules/Mage_All.xml`. It then parses the page-specific layout handles, again looking in all of the layout files for that handle, and finalizes the building of a store page.

Magento's rendering system is built this way to allow seamless addition and removal of modules without affecting other modules in the system. This means you can add, remove and move most of Magento's functionality by simply adding, removing or moving the block declarations in the layout files.

ANATOMY OF A MAGENTO LAYOUT FILE

Layouts contain a small set of XML tags that act as detailed instructions to the application on how to build a page, what to build it with and the behavior of each building block. Here are some behavioral properties of each layout XML tag.

HANDLES

Handle (diagram 1) is an identifier by which the application determines what to do with the updates nested by it.

If the name of the handle is `<default>`, then the application knows that its nested updates must be loaded on almost all the pages of the store prior to loading page-specific layout ("almost all" because some exceptional pages like the product image popup do not load the layout in the `<default>` handle).

If Magento finds handles other than `<default>`, it will assign the updates nested inside the handle to the according page specified by the handle. For instance, `<catalog_product_view>` contains the layout updates for the Product View page, while `<catalog_product_compare_index>` contains those for the Compare Product page. Handles are set-in-stone identifiers that a designer, with no extensive understanding of Magento programming, should never need to modify.

<BLOCK>

Magento determines the behavior and visual representation of each building block of a page via the `<block>` tag. We have already mentioned the two types of blocks Magento employs - structural blocks and content blocks. The best way to distinguish between the two is by examining the behavior assigned to each via the tag attributes. A structural block usually contains the attribute `"as,"` through which the application is able to communicate with the designated area (using the `getChildHtml` method) in a template. You will notice many occurrences of this `"as"` attribute in the default layout, because by nature the default layout is one that builds the groundwork upon which the page-specific layouts can begin adding onto. For instance, in the default layout, there are structural blocks such as `"left," "right," "content"` and `"footer"` being introduced. Although these blocks can exist in normal layout updates, why not first set up the reoccurring structural blocks in the default layout first, and then start adding content on a per-page basis. The available attributes for `<block>` are:

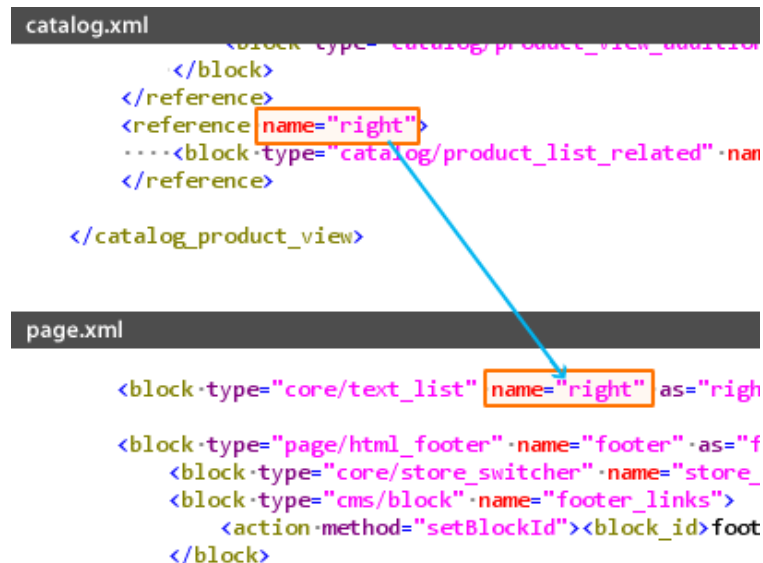
- **type** – This is the identifier of the module class that defines the functionality of the block. This attribute must not be modified.
- **name** – This is the name by which other blocks can make reference to the block in which this attribute is assigned (see diagram 3).
- **before** (and) **after** – These are two ways to position a content block within a structural block. `before="-"` and `after="-"` are commands used to position the block accordingly at the very top or very bottom of a structural block.
- **template** - This attribute determines the template that will represent the functionality of the block in which this attribute is assigned. For instance, if this attribute is assigned `'catalog/category/view.html'`, the application will load the `'app/design/frontend/template/catalog/category/view.html'` template file.

- **action** – <action> is used to control store-front functionalities such as loading or unloading of a JavaScript. A full list of action methods will soon become available, but in the mean time the best way to learn about the different action methods is by playing around with them in the currently available layout updates.
- **as** – This is the name by which a template calls the block in which this attribute is assigned. When you see the getChildHtml('block_name') PHP method called from a template, it is referring to the block whose attribute "as" is assigned the name 'block_name'. (i.e. The method <?=\$this->getChildHtml('header')?> in the a skeleton template correlates to <block as="header">).

<REFERENCE>

<reference> is used to make reference to another block. By making a reference to another block, the updates inside <reference> will apply to the <block> to which it correlates (see diagram 3).

To make the reference, you must target the reference to a block by using the "name" attribute. This attribute targets the <block> tag's "name" attribute. Therefore, if you were to make a reference by <reference name="right">, you are targeting the block named <block name="right">.



```

catalog.xml
<block type="catalog/product_view_additional" name="right">
</block>
</reference>
<reference name="right">
...<block type="catalog/product_list_related" name="right">
</reference>
</catalog_product_view>

page.xml
<block type="core/text_list" name="right" as="right">
<block type="page/html_footer" name="footer" as="f">
<block type="core/store_switcher" name="store_switcher">
<block type="cms/block" name="footer_links">
<action method="setBlockId"><block_id>foot
</block>

```

Figure 29. Code sample from layout files showing use of <reference>.

RULES OF XML

The only set rule you need to remember with regard to XML is that when a tag opens, it must either be followed by a closing tag(<xml_tag></xml_tag>) or self-close(<xml_tag/>) exactly as required by xHTML file tags.

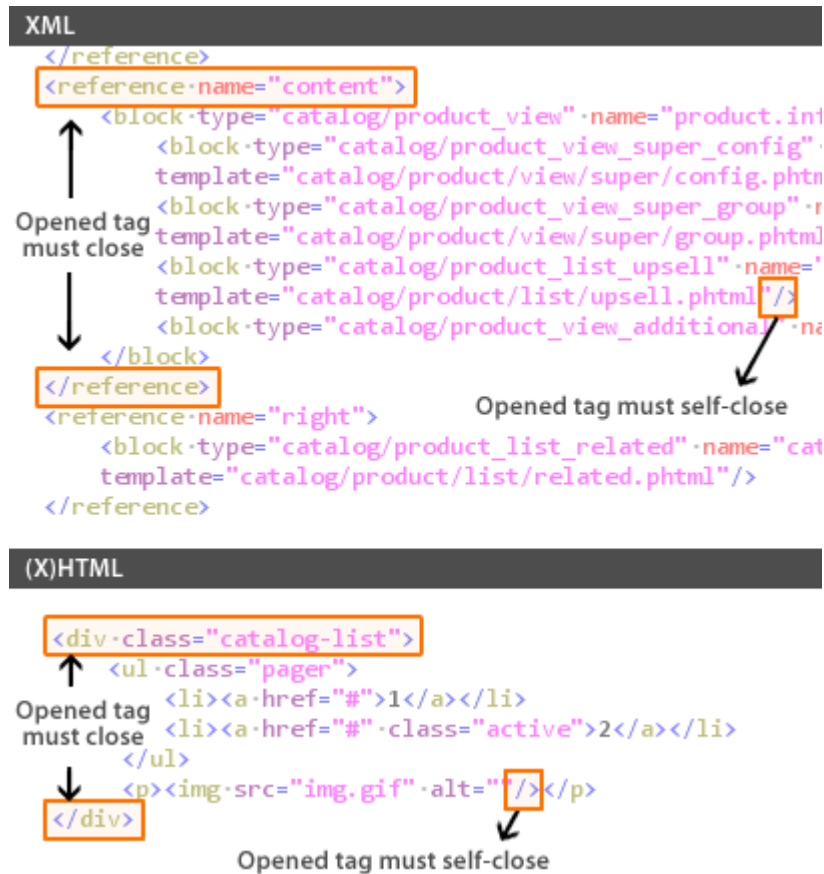


Figure 30. Code sample from layout files showing proper XML syntax.

HOW TO FIND WHICH LAYOUT FILE TO MODIFY

To access the layout files, go to `app/design/frontend/design_package/theme_variation/layout/`. Just like the templates, layouts are saved on a per-module basis therefore you can easily locate the layout file to modify with the help of the Template Hints.

To enable Template Hints, go to **System** → **Configuration** → **Advanced** → **Developer**. At this point you will need to make sure you have your website selected in the **Current Configuration Scope** dropdown instead of **Default Config**. Once your website is selected in the dropdown, click **Debug**. Here, you can change **Template Path Hints** to **Yes**. If you want even further information, you can also change **Add Block Names to Hint** to **Yes**.

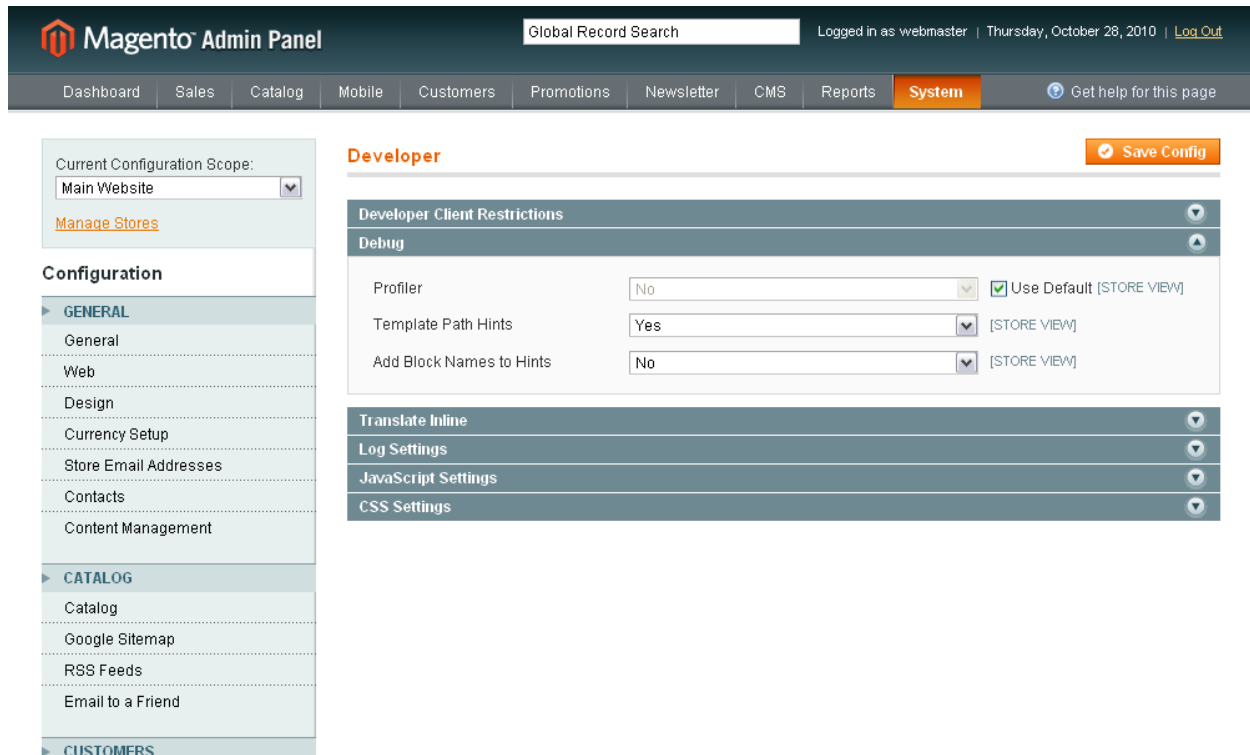


Figure 31. xxx.

Now that you have Template Hints enabled, reload the page you want to modify, and look at the path of the template file(s) that the Template Hints will provide you with. If you want to (for instance) move the mini cart, reference the template path (ex:app/design/frontend/base/default/template/**checkout**/cart/sidebar.phtml) and use the first folder name inside the theme folder (indicated in bold, which is the module name) to find the according layout file. Therefore, in the case of the mini cart, you know to look in "**checkout.xml**" to modify the mini cart positioning. Each layout file is further broken into handles, which specify layouts for specific page types. Each area of per-page layout is clearly marked with comments reflecting its usage, but the application itself recognizes the layout separation by the handles of each layout.

QUICK EXERCISES TO GET YOU STARTED

Here are a couple of exercises for you to get a feel for working with layouts. These examples are based on the Community Edition. In these exercises, you will be copying layout files from the base package to your custom package and modifying them there. We will explore other methods that do not require you to copy the entire file later.

EXERCISE #1

In the category page, move the "My Cart" box from the right column to the left

1. Turn on the Template Path Hint by going to the admin then navigating to System → Configuration. When you are in the configuration page, select the store you are working in by using the top left

website/store selector. Wait for the page to reload, and then select the Developer tab. Select "Yes" in the select box for Template Path Hints. Click Save. Go back to the store front, and reload.

2. When the page reloads, look at the template path of the "My Cart" block- it will most likely say "frontend/base/default/template/checkout/cart/sidebar.phtml." By looking at the path, you know this template is being introduced via the "checkout" module. How do you know this? - frontend/base/default/template/**checkout**/cart/sidebar.phtml. It says so in the template path. The immediate directory name following "template" is the name of the module through which a template is introduced.
3. Because now we know we're dealing with the checkout module, copy the checkout.xml file from the base package to your custom design package.
Copy app/design/frontend/base/default/layout/checkout.xml to
app/design/frontend/<your_package>/default/layout/checkout.xml
4. Open the copied file app/design/frontend/<your_package>/default/layout/checkout.xml - Search for "checkout/cart/sidebar.phtml" (the template name of the My Cart block) in the layout updates. You will find an area that looks like this:

```
<reference name="right">
<block type="checkout/cart_sidebar" name="cart_sidebar" before="-" template="checkout/cart/sidebar.phtml"/>
</reference>
```


Change it to say the following instead (Note that all you are doing is changing the <reference name="right"> to <reference name="left">).

```
<reference name="left">
<block type="checkout/cart_sidebar" name="cart_sidebar" before="-" template="checkout/cart/sidebar.phtml"/>
</reference>
```
5. Reload the store front and you will see the change reflected.

EXERCISE #2

Use the same method to identify and remove or move other content blocks

1. Using the template path hints, identify with Magento module is controlling other blocks you'd like to move or remove (examples include the dog, the poll, the newsletter sign up, etc.)
2. Copy the layout file that controls that layout from the base package to your custom package.
3. Modify the layout file in your custom package to reflect the change you'd like to make. (Comment out or delete blocks you do not want, target them to left, right or footer instead, etc.) Save your changes.
4. Reload the frontend store to see your changes.

CUSTOMIZING USING A LOCAL LAYOUT FILE (LOCAL.XML)

As we have seen above, you can dramatically change what blocks and content Magento shows within a theme by modifying only a few files and including only the modified files in your custom theme. This works because Magento CE v1.4+ and EE v1.8+ use a theme fallback mechanism in which the application looks for each theme file or block declaration first in your custom theme package and then, if it does not find it there, it looks for it in the base theme package. Your custom theme only needs to contain the files, or really just the blocks, that are different from the base theme.

You can use the method of changing and saving files for all versions of Magento and it is a perfectly acceptable way to customize a theme. A more elegant and maintainable approach for turning blocks on and off in Magento CE v1.4+ and EE/PE v1.8+ that allows you to do much of this using just one file is to create a layout file in your custom theme directory that overrides the behavior of only certain blocks in the base theme. You do not need to copy over all the theme files and delete or comment out code in each. You can rely on the base theme to contain all the default blocks and in your theme you simply de-reference the blocks you do not want. In addition, you can do this all in one place so that your changes are easier to keep track of. Built into Magento's logic is that after it has pulled together all of the XML layouts and layout updates, it will pull in and process a local layout file, named "local.xml," last, which can be used as a final override on all previous layout updates for a theme.

To do this, inside of your design package directory, in
app/design/frontend/<your_package>/<your_theme>/layout :

Create a file named local.xml. Inside of local.xml, create a <default> block that will contain and consolidate your global changes:

```
<?xml version="1.0" ?>
<layout>
  <default>
    <!-- Your block overrides will go here -->
  </default>
</layout>
```

Depending on what you want to turn off, local.xml might contain some of the following lines.

```
<default>
  <remove name="left.permanent.callout" /> <!--the dog-->
  <remove name="right.permanent.callout" /> <!--back to school-->
  <remove name="catalog.compare.sidebar" /> <!--product compare-->
  <remove name="paypal.partner.right.logo" /> <!--paypal logo-->
  <remove name="cart_sidebar" /> <!--cart sidebar-->
  <remove name="left.reports.product.viewed" /> <!--recently viewed prod-->
  <remove name="right.reports.product.viewed" /> <!--recently viewed prod-->
  <remove name="right.reports.product.compared" /> <!--
recently compared prod-->
</default>
```

This will vary depending on the theme so this is only guidance. Use the list in the previous section as a guide, find the xml file in your theme's layout directory (or the base layout directory) that contains the block(s) you want to disable and find the names of the blocks in that file. You can then use those names in your local.xml file to remove the blocks. Also, the example above removes content blocks from the default

scope; you may prefer to remove them only from specific structural blocks—like "left" or "right." But this gives you an idea of how it works.

You can also use `local.xml` to update specific handles. For example, if you wanted to set the default page template for the category pages to the 2column-left template, you could use the handles that control the rendering of the catalog pages and include code like the following in your `local.xml` file.

```
<catalog_category_default>
<reference name="root">
    <action method="setTemplate"><template>page/2columns-left.phtml</template></action>
</reference>
</catalog_category_default>
<catalog_category_layered>
<reference name="root">
    <action method="setTemplate"><template>page/2columns-left.phtml</template></action>
</reference>
</catalog_category_layered>
```

We have just scratched the surface on using a `local.xml` file to override the layouts for just the blocks we wanted to disable, rather than copying files and deleting code. Using this same approach, you can also add blocks, change where they appear, change some of their behavior, etc., just by overriding the default behavior in a single location and never editing the actual layout files.

CUSTOMIZING USING TEMPLATES

You have seen that the layout files in Magento control the presence or absence of each content block in a theme. What, specifically, gets rendered inside of that block is controlled by the template files. Most templates do not contain any logic about whether they will or will not be rendered—remember that’s typically handled by the layout files. Once a template is called, it is expected that it will be parsed and displayed.

Template files in Magento are PHTML files that contain xHTML and PHP that will be parsed and then rendered by the browser. Once you identify which template file is being used to generate the contents of a specific block, you can modify that template file if desired. Or you can modify the layout file to associate a different template file with a block, which allows you to create completely new template files.

As a general recommendation, rather than editing Magento’s template files, it is a better practice to copy a template file that you want to alter to your theme with a slightly different name and to edit that newly renamed file. Then you would update the layout to reflect that it should use your new template for that block. That preserves the original Magento template in case it is used in multiple places and ensures that when the Magento application is upgraded any changes to the original template file won’t impact your modified template without your explicit review.

Building on the previous example of using a `local.xml` file to override layout declarations, if you wanted your category pages to have a unique page template, you might want to create a new page template in `app/design/frontend/<your_package>/default/template/page/` named “2columns-custom.phtml.” In it you perhaps create a modified version of Magento’s two-column with a left sidebar page template to also include

new structural blocks for cross promotions and free shipping messages that you only want to appear on the category pages.

In this case, your `local.xml` file in your `app/design/frontend/<your_package>/default/layout` directory might include the following code telling Magento to render those pages using your new template file:

```
<?xml version="1.0" ?>
<layout>
<catalog_category_default>
<reference name="root">
    <action method="setTemplate"><template>page/2columns-custom.phtml</template></action>
</reference>
</catalog_category_default>
    <catalog_category_layered>
<reference name="root">
    <action method="setTemplate"><template>page/2columns-custom.phtml</template></action>
</reference>
</catalog_category_layered>
</layout>
```

EXERCISE #1

Separate the SEO links at the footer area - Instead of having the link items to be one list, isolate 'Advanced Search' to be in the header instead.

Which module and layout file assigns a content block to a page is sometimes a little harder to find at first, especially if that block appears on every page in the header or the footer. But you can make a calculated guess that the SEO links are likely controlled by the `catalog` or `catalogsearch` modules or you can see that they are part of the footer structural block and open up `layout/page.xml` and look through the list of children under the footer block in order to locate a block that calls the footer links - you will find `<block name="footer_links">`, which is what calls the SEO links. Now that you know that layout updates reference the SEO links via the `name="footer_links,"` now you will do a search in all the xml files for `<reference name="footer_links">`. You will find references for the `footer_links` block in `catalog.xml` (which calls "Site Map"), `catalogsearch.xml` (which calls "Search Terms" and "Advanced Search") and `contacts.xml` (which calls "Contact Us").

1. Whatever method you use, once you have located the area of the individual SEO link items, you will now begin the steps to isolate "Advanced Search" from the bunch and make it its own thing in the header.
2. Copy any files you will need to modify over to your custom theme.
3. First go to your copied `page.xml` and create a new block
`<block type="page/template_links" name="header_links" as="header_links"`
`template="page/template/links.phtml"/>`
 and nest it inside `<block name="header">`. This makes the layout updates telling Magento to expect this link in `header.phtml`.

4. In the case of the header.phtml file, Magento is hard-coded to look for that filename, so let's just edit the template file directly in our custom theme for this example. Open your template/page/html/header.phtml, and type in `<?=$this->getChildHtml('header_links')?>` where you want the link to reside.
5. Now go to catalogsearch.xml, and cut or comment out this code:

```
<action method="addLink" translate="label title" module="catalogsearch"><label>Advanced Search</label><url helper="catalogsearch/getAdvancedSearchUrl" /><title>Advanced Search</title></action>
```

from `<reference name="footer_links">`. This will remove the link from the footer.
6. Still in catalogsearch.xml, create a new reference to the new header_links block and nest the cut out code inside it like so:

```
<reference name="header_links">  
<action method="addLink" translate="label title" module="catalogsearch"><label>Advanced Search</label><url helper="catalogsearch/getAdvancedSearchUrl" /><title>Advanced Search</title></action>  
</reference>
```
7. Reload the frontend store to see your changes. We now have Advanced search in the header instead of the footer.

6 QUICK GUIDE TO BUILDING A THEME FROM SCRATCH

Many Magento themes, even the commercially available ones, are built in the way described in the previous chapter—by modifying files or overriding blocks that are all defined in Magento’s base package (or base plus enterprise or pro packages for EE and PE).

It is possible though to also create a new Magento theme from scratch and not use Magento’s default structural and content blocks—you are not limited to header, footer, left, right and content as your structural containers if you do not want to be. If you decide to build a theme from scratch, in part or in total, you do take on a lot of added complexity, but depending on your project or your workflow, it may be the right approach for you.

This chapter provides a little bit of guidance around how to breakdown your design and start rebuilding new Magento layouts, templates and skins for it.

ONE: DISABLE YOUR SYSTEM CACHE

To prepare your Magento for production, you need to first disable system cache by going to the Administration Panel (<http://yourhost.com/admin>) and navigating to System → Cache Management. Select the check boxes next to Layouts, Blocks HTML output and Translations and then under Actions select “Disable” and click Submit. Each of those Cache Types should have a red bar in the status area that reads “DISABLED.” By doing this, it will help to ensure that you see a faithful reflection of your store front as you make the changes.

Note: Depending on what you are attempting to achieve, you may need to disable additional cache files in this area or even manually delete cache files in your folder structure. Do not manually delete folders unless you know what you are doing.

TWO: DETERMINE ALL THE POSSIBILITIES OF STRUCTURE TYPES FOR YOUR STORE

Before you even start creating the markup for the store, you will first need to ask yourself the type of page structure you’d like to have for each of your store pages. Make yourself a list that looks something like this:

- Home page will use the three column structure.
- Category listing page will use the two column structure that includes a right column.
- Customer pages will use the **TWO** column structure that includes a left column.

SKELETON TEMPLATE

Once your list is complete, you will create the xHTML markups for each structure type and save them as **SKELETON TEMPLATES** into `app/design/frontend/design_package/theme_variation/template/page/`. A skeleton template is named such due to the nature of its purpose—all it really contains (aside from the `<head>` elements), is the presentational markups that serve to position each **STRUCTURAL BLOCK** into according markup areas.

Sample skeleton template

```
<html>
<head></head>
<body>
<div class="header"><?=$this->getChildHtml('header')?></div>
<div class="middle">
    <div class="col-left"><?=$this->getChildHtml('left')?></div>
    <div class="col-main"><?=$this->getChildHtml('content')?></div>
</div>
<div class="footer"><?=$this->getChildHtml('footer')?></div>
</body>
</html>
```

Upon scanning through the sample skeleton template above, you will notice a PHP method named “<?=\$this->getChildHtml()?>” inside each presentational markup. This is the way Magento loads structural blocks into skeleton templates and hence is able to position all the contents of the structural blocks within a store page.

Each getChildHtml calls on a name as in <div class="header"><?=\$this->getChildHtml('header')?></div>, and these names are ways by which each structural block is identified in the layout. Skeleton templates are assigned to the store through the layout.

THREE: CUT UP YOUR XHTML ACCORDING TO FUNCTIONALITY

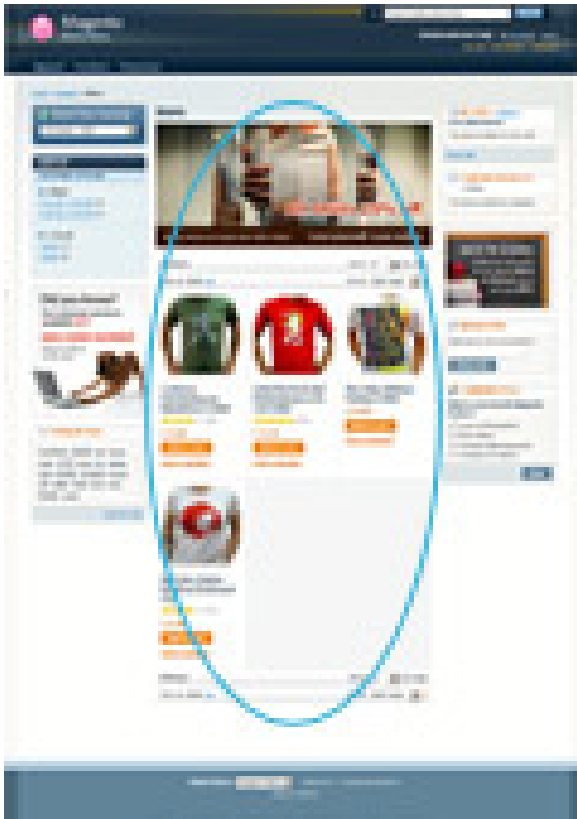
Once you have created your skeleton templates, you will now need to create the template for each content block.

MAGENTO LIKES MEANINGFUL TEMPLATES

You will need to cut up the XHTML markup built for your page and supply the according markup for all functionality of the page. For instance, if you have a mini-cart area in your design, the markup for this area alone will become its own template file. Same for your product tags, newsletter sign up, etc. All of these functionalities already come with Magento, so you can refer to the existing template tags to build your markup logic.

WHERE TO SAVE THE TEMPLATES

Diagram 5



The full markup for any page in your store is introduced via an array of templates that each represents a functionality of a module. In order to find out what templates are being called to a page you'd like to modify, you can turn on the Template Path hints. In order to enable it:

1. Go to the admin and navigate to System → Configuration
2. Select your store in the top left website/store selector
3. When the page reloads, select the 'Developer' tab and select 'Yes' for Template Path Hints.
4. When you are done, go back to the store front, reload your page and you will see the path to all the templates listed according to the block. In order to modify the markup, all you have to do is follow the path written out for you and modify the according template(s).

FOUR: CHANGE THE LAYOUT TO REFLECT YOUR DESIGN

Once you have distributed some of the markups, you probably would like to now move the templates around in a page to see how you are progressing along.

DEFAULT LAYOUT VERSUS LAYOUT UPDATES

There are two types of layouts—default and updates. A default layout (page.xml) is the layout that by default applies itself to almost every page in the store. All other layout files are Layout Updates that simply updates the default layout on a per-page basis.

Let's take for example your skeleton template:

In the default layout, you have it set to three columns, which means by default most all of the page in your store will have the three column page structure. But it is not the three column structure you need for your product page. For IT, you want a two-column structure that includes a right column. To accommodate this, you will leave the default layout alone and open catalog.xml in which you can place some layout commands that tells the application to load the two-column structure to your product page instead of the default three. This is called the process of updating a layout.

Example method of assigning skeleton template:

```
<reference name="root">
  <action method="setTemplate"><template>page/2columns-right.phtml</template></action>
</reference>
```

Let's take another example:

Say by default that you want a newsletter sign-up box in your right column, but in customer account pages you want to exclude it. In this case, you would leave your default layout alone and open up customer.xml, into which you will place a command that unsets the newsletter content block, excluding the newsletter functionality from the page.

7 OTHER RESOURCES

Magento offers a wealth of resources to help you in your site design. We encourage you to use and participate in the following resources:

- [Knowledge base articles](#)
- [Webinars](#)
- [Forums](#)

We also invite you to share with us your designs, discuss this documentation and ask lots of questions at the community forum's [‘HTML, XHTML, CSS, Design Questions’](#) thread.

